University of Zurich[UZH]

Department of Informatics

A New Approach to Product Line Engineering in Model-Based Requirements Engineering

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zurich

for the degree of
Doctor of Science

by
Reinhard Stoiber
from Austria

Accepted on the recommendation of
Prof. Dr. Martin Glinz
Prof. Dr. Krzysztof Czarnecki

2012

University of Zurich[UZH]

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, April 4, 2012*

Head of the Ph.D. committee for informatics: Prof. Abraham Bernstein, Ph.D.

* Graduation date

Acknowledgments

Although an acknowledgments section is not mandatory in a doctoral thesis I would like to use this opportunity and say thank you to those who helped and supported me during my years working on this thesis.

First of all, I would like to thank Professor Martin Glinz (University of Zurich) very much for hiring me into the Requirements Engineering Research Group and for providing a stimulating working environment in this group. He always gave me the freedom to explore new ideas and to delve into them in various projects with students and colleagues. His feedback and advice has been very substantiated and elaborate. Without him this thesis would not have been possible.
Secondly, I would also like to thank Professor Krzysztof Czarnecki (University of Waterloo) for earlier constructive feedback at conferences and meetings and for agreeing to step in as a co-advisor for this thesis.

Further, I am especially grateful to my colleagues in the requirements engineering (RE) research group for the countless conversations about RE-related and non-RE-related topics, for keeping their attention whenever I needed to test new ideas for initial commonsense judgement and for the social activities, sports activities and fun during coffee breaks and while playing tabletop soccer in the department's basement.

With regards to the content presented in this thesis, I am particularly obliged to thank Ivo Vigan (ETH and CUNY) and Cédric Jeanneret (UZH) for their plentiful and very qualified feedback. Ivo has been a programmer in our group and helped me a lot to clarify many of the details of the presented work and in particular to understand Boolean satisfiability solving in this context. And Cédric has helped me a lot with many things in this thesis. We had countless discussions about modeling fundamentals, model composition and the nature and contribution of this thesis. This has improved and shaped the presented work considerably.

There were many students involved in this research as well and they also made a significant contribution to the presented work. Most importantly these include Anil Kandrical (UZH), Michael Jehle (UZH), and Stefan Schadauer (JKU Linz), who did their master's theses, and Urs Zoller (UZH), who did his bachelor's thesis in the context of my work.

I also want to generally mention all the people I met at conferences, workshops, working groups and other meetings around the world. The discussions we had have often triggered smaller or bigger learning processes that may also have influenced some of the descriptions and views presented in this thesis.

For proof-reading, editing and polishing my English I am further grateful to Thomas Armstrong (Language Center UZH and ETH Zurich), Bobby Rohrkemper (ZHAW Winterthur), Eunsuk Kang (MIT CSAIL), and Shamal Faily (University of Oxford). They have provided several important presentation improvements and comments that helped me to finalize this thesis.

Finally, of course, I want to thank my parents, relatives, and friends, for their continuous mental support and encouragement during all those years I was working on this Ph.D. thesis.

Abstract

Variability modeling is a major issue in requirements engineering for software product lines. Existing state-of-the-art approaches heavily leverage the principle of separation of concerns by specifying the requirements model in multiple separate diagrams (for example, the UML) and using orthogonal variability modeling to describe commonality and variability (for example, feature models). Mapping-based approaches are further used to accurately specify the variability's impact on the requirements model. However, this makes variability-related requirements engineering activities unnecessarily cumbersome, since the specification of variable features is not handled as a primary concern, but rather scattered over many separate diagrams.

This thesis presents a new approach that builds on different premises, which neither require variability mappings nor an orthogonal variability model. Instead of separating the requirements model into multiple diagrams, a fully *integrated* requirements modeling language and tool support for *view generation* are used. Instead of using an orthogonal variability model and a specific variability mapping approach, a *compositional* approach is used. On this basis a new, fully-fledged product line requirements modeling approach has been developed. The approach is parsimonious in the sense that it aims at extending an existing language as little as possible. It also allows a fine-grained specification of cross-cutting features and their functional dependencies, when needed. It allows the requirements and variability model to be visualized in a single view and more abstract views at arbitrary levels of abstraction to be generated. It provides novel support for product derivation that can visualize both a decision's impact on the product's functionality and on other variability decisions in the same diagram and tool. It continuously verifies the satisfiability of the model and allows advanced automated analyses such as constraint propagation, based on Boolean satisfiability (SAT) solving. In addition it provides advanced support for variability model creation and evolution by allowing a straightforward, semi-automated extraction and composition of any identified variable feature.

The empirical validation presented in this thesis is four-fold. First, a constructive tool implementation proves technical feasibility. Second, the modeling of several real-world examples along with state-of-the-art solutions shows practical feasibility. Third, a rigorous performance analysis verifies that SAT solving scales well for models of this new type, which proves that the presented automated variability analysis solution is feasible. And fourth, a recent real-world case study compares the practical performance of the presented approach with an industry-strength and state-of-the-art solution and shows considerable benefits of the presented approach.

This thesis contributes a complete description of this new approach, which is mainly based on the ADORA requirements and architecture modeling language. The presented approach is of a general nature, however, and we expect our empirical results to yield equally encouraging data also with any other language that satisfies the approach's prerequisites. We hope that this integrated approach to requirements modeling (or conceptual modeling in general) and variability modeling will soon also be applied with other modeling languages. The presented work will possibly lead to an emergence of new types of tools that can also visualize existing product line models (for example, specified with UML and feature modeling) in a new, integrated and flexible manner, as presented in this thesis. This could profoundly change and improve the way engineers and analysts visualize and deal with variability in software models in the future.

Zusammenfassung

Variabilitätsmodellierung ist eine wichtige Angelegenheit in der Anforderungsanalyse von Software Produktlinien. Existierende Ansätze mit heutigem Stand der Technik stützen sich stark auf eine Separierung der Anforderungsmodellierung in verschiedene Diagramme (typischerweise unter Verwendung von UML) und benützen orthogonale Variabilitätsmodellierung zur Beschreibung von Gemeinsamkeiten und Variabilität (zum Beispiel Feature Modelle). Zuordnungs-basierte Ansätze werden weiters verwendet um die Variabilität in der Anforderungsmodellierung präzise zu spezifizieren. Dies macht Aktivitäten im Zusammenhang mit Variabilität in der Anforderungsanalyse unnötig umständlich, da die Spezifikation der Variabilität nicht als grundlegendes Element realisiert ist, sondern vielmehr verteilt über verschiedene Diagramme beschrieben wird.

Diese Dissertation präsentiert einen neuartigen Ansatz, der weder Zuordnungen noch ein zusätzliches orthogonales Variabilitätsmodell benötigt. Anstatt das Anforderungsmodell auf mehrere Diagramme zu verteilen, wird eine vollständig *integrierte* visuelle Anforderungsmodellierungssprache zusammen mit Werkzeugunterstützung zur *Sichtengenerierung* verwendet. Und anstatt ein orthogonales Variablitätsmodell zusammen mit einem Zuordnungs-basierten Ansatz zu verwenden, wird ein *kompositionaler* Ansatz eingesetzt. Auf dieser Basis wurde ein neuartiger und vollständig ausgeprägter Ansatz zur Anforderungsmodellierung von Software Produktlinien entwickelt. Der Ansatz ist sparsam, da er darauf zielt mit einer kleinstmöglichen Erweiterung einer bestehenden Sprache auszukommen. Dennoch erlaubt er eine sehr genaue Spezifikation von querschneidenden variablen Features und deren funktionalen Abhängigkeiten, wenn dies gebraucht wird. Er erlaubt eine integrierte Darstellung des Anforderungs- und Variabilitätsmodells und die Generierung von abstrakten Sichten auf beliebigen Abstraktionsstufen. Weiters erlaubt er eine neuartige Herangehensweise zur Produktableitung, welche die Auswirkungen einer Entscheidung sowohl auf die Funktionalität des Produkts als auch auf die

weiteren Entscheidungen im gleichen Diagramm und Werkzeug visualisieren kann. Die logische Erfüllbarkeit des Modells wird laufend geprüft, was auch eine breite Palette an automatisierten Analyseoperationen erlaubt, die auf Boolescher Erfüllbarkeitsauswertung (SAT) aufbauen und zum Beispiel die automatische Propagierung der Auswirkung von bestimmten Entscheidungen auf die restlichen Entscheidungen erlaubt. Der Ansatz erlaubt weiters auch eine fortgeschrittene Art der Erstellung und Weiterentwicklung von Variabilitätsmodellen durch einfache, semi-automatische Extraktion und Komposition ausgewählter variabler Features.

Die empirische Validierung dieser Dissertation gliedert sich in vier Teile. Zuerst beweist eine konstruktive Werkzeugimplementierung die technische Machbarkeit des Ansatzes. Als Zweites zeigt die Modellierung von realen Beispielen im Vergleich mit existierenden Lösungen am Stand der Technik die praktische Tauglichkeit des Ansatzes. Als Drittes zeigt ein rigoroser Leistungstest, dass die präsentierte automatisierte Analyse des Variabilitätsmodells gut skaliert und ebenso praxistauglich ist. Und als Viertes zeigt eine aktuelle Fallstudie, dass der praktische Einsatz des präsentierten Ansatzes im Vergleich zu führenden Werkzeugen mit aktuellem Stand der Technik auch bedeutende Vorteile birgt.

Diese Arbeit präsentiert eine vollständige Beschreibung dieses neuen Ansatzes, welcher hauptsächlich basierend auf der Anforderungs- und Architekturmodellierungssprache ADORA beschrieben wird. Der Ansatz ist jedoch von genereller Natur und wir erwarten ähnlich vorteilhafte Ergebnisse auch für andere Sprachen, die alle nötigen Voraussetzungen für diesen Ansatz erfüllen. Wir hoffen, dass dieser integrierte Ansatz zur Anforderungsmodellierung (bzw. zur konzeptuellen Modellierung im Allgemeinen) und Variabilitätsmodellierung auch bald in anderen Modellierungssprachen zum Einsatz kommt. Die präsentierte Arbeit wird möglicherweise zu neuartigen Typen von Werkzeugen führen, welche auch existierende Modelle von Software Produktlinien (welche zum Beispiel mit UML und Feature Modellierung spezifiziert sind) auf eine neue, integrierte und flexible Art und Weise visualisieren können, wie in dieser Dissertation präsentiert wird. Dies könnte die Art und Weise wie Ingenieure und Analysten in Zukunft Variabilität in Software Modellen visualisieren und behandeln grundlegend verändern und verbessern.

Contents

List of Figures

List of Tables

Part I

Motivation, State of the Art, and Open Challenges

CHAPTER 1

Introduction

1.1 Introduction and Motivation

A reasonable specification of requirements plays an important role in any successful software project. Without specifying a software system's key requirements adequately and precisely enough there remains a considerable risk of developing software that is not fit for its purpose. Rectifying misunderstandings of requirements later in the software development process can result in major additional cost and delay [Boehm, 1976] [Glinz, 2006].

requirements engineering

Requirements Engineering (RE) is the discipline to elicit, analyze, document, and validate the stakeholders' requirements for a planned system [Sommerville and Sawyer, 1997] [Pohl, 2010]. While RE is an activity that introduces additional costs to the software development life-cycle, its benefits (e.g., higher quality, higher product acceptance rates, and errors are found earlier and can be fixed while it is still relatively cheap) typically outweigh these extra costs [Glinz, 2006], as follows.

Studies have found that for very large percentages of errors in software products, their origin can actually be traced back to the requirements phase. Endres found that 46 percent of all errors of an operating system had their origin in poor understanding and communication of the problem [Endres, 1975]. Sheldon et al. studied the software development life cycle of Air Force systems and found that overall 41 percent of errors could be traced back to requirements [Sheldon et al., 1992]. Hall et al. reported that 48 percent of all problems experienced in software development

projects in twelve companies in different domains were requirements-related [Hall et al., 2002]. Boehm showed that the cost of rectifying errors made early in the development process is profoundly higher than the cost of rectifying those that were made late [Boehm, 1981]. Boehm and Basili argued that finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it already during the requirements and design phase [Boehm and Basili, 2001]. Therefore, requirements engineering should be a major concern in most software development projects. Not performing a minimum degree of requirements activity would be irresponsible and makes projects likely to result in low quality and increased cost. On the other hand, exaggerating and performing too much requirements activity can be as dangerous as too little requirements engineering, by making the project late and over-budget [Davis, 2005]. Hence, a good balance needs to be found with the goal of performing just enough requirements engineering for every software project.

requirements modeling

Requirements need to be documented in order to efficiently analyze, communicate, and validate them. In 1993 the Institute of Electrical and Electronics Engineers (IEEE) introduced a standard to specify a software requirements specification as a textual and structured document, of which a revised version was published in 1998 [IEEE, 1998]. A few years ago many authors argued that the days of large, word-processed requirements documents were over, though [Davis, 2005]. Stating single requirements, categorizing them, organizing them in tables, and performing advanced analysis on them increasingly became state-of-the-practice, because it made software projects much more manageable. Recent research, however, has gone further and has introduced modeling for a more systematic software development and to support requirements documentation and specification. In software engineering, the Unified Modeling Language (UML) [Rumbaugh et al., 2004] has become quite popular. But also many other modeling languages and notations have been proposed to improve the state of the art of requirements engineering, see, e.g., [Davis, 2005], [Glinz, 2006], [van Lamsweerde, 2009], or [Pohl, 2010]. In terms of visualization and analysis, modeling has major advantages compared to only specifying and managing lists and tables of textual requirements. Models can describe functional requirements simpler and clearer, can better support hierarchical decomposition and often make it easier to derive a corresponding implementation, compared to purely textual specifications. However, additional training may be necessary to ensure a correct use of these modeling languages. While the vast majority of requirements specifications created today are still written in natural language, augmented with tables, pictures, and diagrams, there is a general trend towards relying more on (lightweight) modeling [Glinz, 2010b].

software product lines

Whenever a company finds itself spending more and more effort to customize existing software systems, instead of freshly developing new-to-the-world software products, software reuse becomes of paramount importance to ensure an efficient development and high-quality software. Not reusing identical and already previously developed functionality will inevitably lead to unnecessary additional cost and typically to lower software quality (reuse raises quality because reused components were previously quality-assured and many bugs are already known or fixed).

Software product lines (SPLs) aim at maximizing the amount of reuse of any previously developed software artifacts for the development of multiple software products. Whenever several existing and/or planned software products share a significant amount of commonality and their differences can easily be thought of as variability, then the adoption of a software product line approach is highly advisable.
Early work on requirements engineering for software product lines was originally called domain analysis [Neighbors, 1980], which describes the system analysis not only for a specific problem, but for a collection of possible problems. In general, David Parnas' work on realizing sets of software programs as program families may have been the first published work that explicitly addressed the main ideas of today's software product line engineering [Parnas, 1976]. Parnas' seminal work was focused on software programs, however, and not yet on software requirements. Nevertheless, as long ago as 1980, James Neighbors already stressed that the key to reusable software is in the reusability of analysis and design, rather than code.
More recently, Clements and Northrop highlighted the particular benefits, case studies and guidelines on how to achieve institutionalization of a product line engineering process [Clements and Northrop, 2001]. They argue that when skillfully implemented, a product line strategy can yield enormous gains in productivity, quality, and time-to-market for software organizations of all types and sizes. Pohl et al. further presented a comprehensive framework for software product line engineering (SPLE) and show how to systematically model variability and link it with any other engineering artifacts throughout the software engineering process [Pohl et al., 2005].

variability in requirements —when?

Explicitly considering variability modeling in requirements engineering is crucial for developing, maintaining and evolving a successful *software product line*. An ideal variability model must on the one hand be fine-grained enough, so that it covers everything a customer may want to configure, and must on the other hand be abstract enough, so that it does so with an overall minimum amount of variable entities. Too much variability not required by any customers and/or markets may again result in a less effective product line. Only when the necessary amount of commonality, variability, and dependencies is well understood and well adjusted to current and future (envisioned) markets, can high-quality and tailored software products be rapidly instantiated at a minimal cost.
Furthermore, variability in requirements specifications may not only make sense when developing software product lines, but also in a wider context. For example, when many stakeholders are involved in the development of a large monolithic system, some requirements may be critical for all stakeholders, but others may be relevant for some particular stakeholders only. When all requirements are documented and need to be *reviewed*, it may be laborious for many stakeholders to receive the full requirements specification, while they are particularly interested in reviewing some key requirements only. Thus, specifying fragments that are only relevant to particular stakeholders or stakeholder groups as variability and using a product line approach to generate tailored specifications is essential. Hence, variability in requirements specifications may also surface in single system development with diverse stakeholders or stakeholder groups with deviating interests.

existing research

Research has addressed the fields of requirements engineering, requirements modeling, variability modeling (e.g., domain engineering) and software product line engineering for more than three decades. This previous work provides valuable fundamental concepts that this thesis[1] builds upon. In requirements modeling, the state of the art today is the use of multiple diagram types to model various facets of a system, which makes large models considerably better scalable for human engineers. The most prominent of these languages is the UML. But also other requirements modeling languages have been developed, which model all facets of a system in a fully integrated diagram and use view generation to make this modeling scale. ADORA, which we describe in Section 2.3, is such a language, for example. In Chapter 2 a general introduction to requirements modeling and these two languages is provided, as far as required for a solid understanding of this thesis. In Chapter 3 the state of the art in variability modeling is summarized, where various languages and approaches have emerged. These variability modeling languages and approaches all introduce a dedicated orthogonal variability modeling notation and build on mapping-based approaches to handle the variability in requirements models. The most widely used notation for variability modeling is feature modeling, for which a wide variety of automated analysis support has also been developed. This automated analysis support allows tools that help identifying and resolving inconsistencies among variability constraints and much more to be developed.

open problems

Despite the considerable previous research efforts in these areas (e.g., requirements modeling, variability modeling, software product line engineering), significant problems in today's state-of-the-art approaches remain. When developing a software product line (or a mere product line of requirements specifications) the specified variable features are certainly of primary importance for the success of the product line as a whole. This importance should adequately be reflected in the design of the used requirements and variability modeling languages and notations. State-of-the-art variability modeling approaches, however, all have one thing in common: they propose an additional, orthogonal variability modeling notation, build on requirements modeling languages that require multiple diagrams to specify different facets of the model (e.g., the UML) and use a mapping-based approach to specify variable model elements in the requirements model. While there is a wide variety of mapping-based approaches to relate variability and requirements models (Section 3.2.2), relying on mappings is the predominant approach taken today. Classic requirements modeling languages like the UML split the requirements model into multiple diagrams of complementary types (e.g., class diagrams, use case diagrams, etc.), which allows a better visual scalability of the model for large single system development, but causes difficulties when variability arises. Every variable feature typically impacts many other facets (i.e., UML diagrams) of a software system and this leads to an information scattering of the variability specification over all these separate diagrams. The precise requirements for every single variable feature are specified by mapping-based approaches (for example, by adding feature annotations in the UML model). In a less rigorous product line development

[1] The term *thesis* is used synonymously with the term *dissertation* in this document. Both refer to this written Ph.D. thesis as a whole.

process the variability model may still be very valuable, though, only to get an overview of the available features. However, a more detailed functional requirements specification of these features, other than only their names and interdependencies, is often required for a clear and unambiguous variability specification. Further, from a requirements viewpoint, specifying variable features only with annotations or similar in other diagrams does not really handle variable features as primary concerns in a product line's requirements specification. Conventional single-system development concerns like components, classes, and use cases are still the primary concerns, while the specification of variable features is a mere annotation of orthogonal properties specified in a separate, orthogonal variability model. This information scattering of the conceptual model over various separate diagrams leads to heightened efforts for specification and maintenance of the variability and requirements model. It makes the comprehension of the impact of any feature selection or deselection difficult to grasp for human engineers because this information is spread over many separate diagrams and can not be visualized with an integrated view. These are fundamental open problems today—see Chapter 4 for more details.

general research goal

The general research goal of this thesis was to develop a new approach to product line requirements modeling that builds on fundamentally different basic assumptions and, thus, completely avoids the occurrence of these open problems. A new approach had to be designed and developed for this purpose, which does not have the characteristics that cause this information scattering. Basic assumptions and premises that are different from state-of-the-art approaches had to be formulated, as presented in Chapter 5.

Further, a solid validation of this new approach had to be a research goal as well. This validity evaluation must show the approach's general feasibility, by realizing it with a tool, for example. It must show whether the approach is expressible enough and feasible to be used with real-world product line examples. For all the provided automated analysis support it must show that the presented solution is feasible and scales reasonably well. And finally, an application of this new approach in a real-world software product line setting must be performed, which compares this new approach's overall performance with today's state-of-the-art solutions. Along with the conceptual development and realization of such a new approach these empirical results are required as well, in order to provide a substantive scientific contribution.

1.2 Contribution

general contribution

The contribution of this thesis is a demonstration of a new type of integrated and compositional approach to modeling a software product line that brings significant benefits and improvements to state-of-the-art solutions. It shows that a fully fledged software product line engineering approach can be built on the principles of *notational integration*, *feature composition* and *view generation* (integration, composition, and view generation, in short, see Chapter 5), rather than on separate diagrams and annotation (e.g., [Czarnecki and Antkiewicz, 2005] or

[Kästner, 2010]). It also shows how this can be done (Chapters 5–10). From a scientific point of view, it provides a comprehensive empirical validation and early evidence of the feasibility and particular benefits of such an approach (Chapters 11–14).

contributions to research and practice

With regards to requirements engineering activities in the software product line engineering process, the major contributions of the presented approach to software engineering research and practice may be the following three:

- **Stepwise, incremental product derivation:** State-of-the-art solutions separate the product configuration and the actual product generation and use multiple separate diagrams to specify the variability and requirements model. Product configuration is typically done on the basis of a feature model, today, while the generation of tailored product requirements specifications is typically performed automatically by the push of a button, once the configuration is complete. Some tools like pure-systems GmbH's pure::variants or Voelter's projectional language workbenches [Voelter, 2010] already allow a live re-generation of a consistent and (partially) derived requirements model every time the feature configuration changes. They do so by filtering the annotated diagrams. However, the impact of such a change is still scattered over multiple separate diagrams, as in all state-of-the-art approaches. The approach presented in this thesis does *not* separate these two, but *integrates* them—in both the language and tool used for requirements and variability modeling. Whenever a variability binding decision is taken (i.e., when a feature is selected or deselected), the constraints are instantly propagated such that the new configuration is again fully valid (Chapters 8 and 9) and the visualized requirements model is instantly transformed such that the newly shown requirements model represents exactly the currently set configuration (i.e., by weaving, removing and/or re-visualizing the variable features whose truth value just changed, see Chapters 7 and 8). Stepwise, incremental product derivation allows visualizing all these effects in a single, integrated view. This way the complete impact of any change of a variability binding decision (e.g., between the values *selected*, *deselected*, and *undecided*) can fully be visualized within a single view, at any time during a product derivation. To enable a fully flexible configuration the presented approach also allows manual changes of automatically propagated decisions, where the emerging conflicts are resolved fully automatically (Chapter 9). Such conflicts are resolved by automatically finding and propagating minimal satisfiable change sets (Section 9.4.2). This automated resolution of configuration conflicts is novel, too, to the best of our knowledge. A remotely similar solution has only recently been introduced for feature models [Xiong et al., 2011]. Overall, stepwise, incremental product derivation allows an easy evaluation of *what-if scenarios*, where engineers and customer-stakeholders can straightforwardly let the tool calculate and visualize the effects of various configuration choices and the trade-offs between different product variants.
- **Simpler transition towards a product line:** Before developing a systematic software product line, companies typically maintain a comprehensive reference specification that includes the functional specification of most or all previously developed functionalities

and features. In state-of-the-art approaches transitioning towards a software product line typically means that an orthogonal variability model is created and this reference specification is annotated with mappings to the orthogonal variability model (Section 3.2.2). Creating and maintaining all these mappings implies rather much manual effort, though, because of the combinatorial explosion when the models grow larger. The approach presented in this thesis allows an integrated and compositional variability modeling, which does not require any mappings. This allows us to develop novel tool support for a semi-automated extraction of variable features, which we call *feature unweaving* (Chapter 10). Feature unweaving is a tool function that automatically extracts a selection of variable model elements, provided by a domain requirements analyst, into a compositionally specified variable feature (e.g., based on aspect-oriented modeling). Because the approach allows visualizing the complete model in an integrated visual notation such a semi-automated decomposition with feature unweaving becomes possible and feasible. Feature unweaving (i) does not require any additional orthogonal variability model for product line requirements engineering, (ii) allows an easier variability specification that is less laborious than with separate diagrams and an annotative approach and (iii) makes the remaining model of the commonality simpler and easier to comprehend with every extraction. While the underlying aspect-oriented modeling may seem complex, this is not an issue, because it is generated automatically. Furthermore, the correctness of any generated extraction can automatically be evaluated and guaranteed by the tool. Overall, the presented approach allows a simpler transition of a reference requirements model into a software product line requirements specification.

- **Correctness by construction:** The presented approach ensures that every created software product line model and every derived product model is *correct by construction*, to a specific extent. This is done by (i) verifying the correctness of every new variability extraction with feature unweaving, (ii) continuously verifying the satisfiability of all specified variability constraints, and (iii) systematically calculating the constraint propagation for every taken variability binding decision so that the variability constraints always remain satisfied. First, when a feature specification is created with feature unweaving, the correctness of the created aspect-oriented modeling is verified automatically. This is done by verifying the semantic equivalence of the original model with a copy of the model where the selected feature was extracted and woven again—if these two models are semantically equivalent then the extraction is correct. Second, every time the underlying variability model changes (e.g., whenever any dependency between any variable entity is changed) the presented approach re-interprets the model at hand in a formal way and verifies its satisfiability and other crucial properties with a Boolean satisfiability (SAT) solver (Chapter 9). This continuous SAT-based automated analysis guarantees that all product line models created using this approach are free of inconsistencies. Third, it also guarantees that every derivable application product specification satisfies all specified variability constraints. Such a SAT-based verification is always feasible because the product line as

a whole is continuously kept satisfiable (i.e., any change in the model that leads to unsatisfiability is immediately undone). With appropriate tool support this solution avoids requirements conflicts already during product line model creation and configuration time, as far as the necessary constraints were accurately specified. Without such tool support, this correctness would need to be verified manually, which is a tedious, laborious and error-prone process. The presented solution continuously and automatically runs this correctness verification in the background. This relieves the engineers of such tasks and they can even rely on the fact that all their created modeling is correct to a significant extent. Overall, the presented approach assures the correctness of every semi-automatically created variable feature specification, of the variability model as a whole (i.e., with regards to its variability constraints), and of all generated product models.

technical contributions

From a more technical and conceptual point of view, the contribution in terms of new characteristics realized in the presented approach are the following:

- We present a *general approach* to integrated software product line modeling that can be realized with any modeling language, despite its original development based on ADORA. The approach is feasible whenever the two prerequisites of *integration* and *composition* are satisfied and appropriate tool support for *view generation* is provided for scalability, as specified in Chapter 5. We use ADORA's integrated concrete syntax and ADORA's existing aspect-oriented modeling capabilities for demonstration in this thesis, though.
- We present an approach that allows a fully *integrated concrete syntax* and visual notation for modeling *requirements and variability* in an integrated and coherent diagram, where no mappings are required. A similarly integrated concrete syntax and visual notation does not exist in today's literature. This contribution could also be summarized as a complete and minimalistic *language extension* to an existing integrated and compositional modeling approach, which allows a fully fledged product line variability modeling. We call this language extension Boolean decision modeling, see Chapter 6.
- We introduce a new solution for *feature weaving* (Chapter 7) that allows an on-the-fly and always consistent product generation already during product derivation and configuration time (Chapter 8). Single variable join relationships (i.e., pointcuts or parts of pointcuts, which are used for variability modeling in the presented approach) can selectively be woven, removed and/or re-visualized in an aspect-oriented form. The visual representation of variability constraints gets consistently derived, too. This allows the generation and visualization of an accurate and consistent requirements model for any partial or full product configuration.
- We allow a direct and straightforward specification of *cross-cutting variable features* (e.g., variable features that impact multiple other variable features). This leads to a more natural variability representation, particularly for heterogeneously cross-cutting variable features, which occur rather frequently in realistic product lines. While aspects have

already previously been used for representing (cross-cutting) variable features, systematically doing so with a fully integrated requirements modeling notation is new. Cross-cutting allows variable features to have multiple parent features and hence also leads to fewer additionally required variability constraints because of technical dependencies. This leads to the need of additional mechanisms for specifying weak hierarchical dependencies, though, as follows.

- The fact that cross-cutting variable features may have multiple parent features sometimes requires the ability to specify *weak hierarchical dependencies*. We introduce such weak hierarchical dependencies, present their detailed formal semantics and show how they need to be considered in the presented SAT-based automated analysis. In existing research, variability models are handled orthogonally and are typically structured in a pure tree format. Hence, a feature always has at most one parent feature and the model's hierarchy constraints require that for any selected feature its parent feature must also be selected. This yields a strong hierarchical dependency. For cross-cutting features an interpretation of every parent feature as strongly required is often inaccurate, though. A brief example for such a case is the *TV* feature in the infotainment system of a real-world automotive product line (Section 5.2). The *TV* feature's functionality extends both the *Navigation System* feature and the *Rear Seats Entertainment* feature. However, it strongly depends only on the *Navigation System* to be realizable (i.e., to function correctly) and does *not* strongly require the *Rear Seats Entertainment* feature. When the *Rear Seats Entertainment* is selected and the *TV* feature is selected too, it extends this feature's functionality. However, when the *Rear Seats Entertainment* is *not* selected, the *TV* feature is still validly selectable (i.e., the child feature can still be selected despite one of its parent features may be deselected). Such a case requires the explicit specification of a *weak hierarchical dependency*. The language fundamentals and a rigorous solution for the automated analysis of such a specification are presented in Sections 6.1.1, 9.1 and 9.3 of this thesis. Such explicit weak dependencies among features are novel and have never explicitly been discussed in existing work, to the best of our knowledge.
- Finally, we present a solution for *SAT-based automated analysis* of variability constraints that goes beyond the state of the art in two regards. First, it can fully deal with both cross-cutting variable features and weak hierarchical dependencies. It also works on the level of decision items, which can be more fine-grained than variable features (Section 6.1.1). Second, it is capable of instantly and automatically resolving any configuration conflicts that may emerge during a product configuration (i.e., a product derivation). Hence, any reachable partial or full product configuration will always lead to a product that satisfies all variability constraints. This approach to product configuration, which guarantees correctness by construction, is also novel. See Chapters 8 and 9 for more details.

1.3 Thesis Outline

Part I The remainder of Part I is structured as follows. Chapter 2 first gives a concise overview on the fundamentals and state of the art in requirements modeling. It particularly focuses on the requirements modeling languages UML and ADORA and on the state of the art in aspect-oriented modeling. Chapter 3 motivates the importance of variability modeling and introduces the state-of-the-art languages and approaches in this area, along with the existing automated analysis approaches and solutions. Both of these chapters particularly focus on those state-of-the-art characteristics that are crucial to understand the contribution of this thesis. Chapter 4 further highlights the three major open problems in all these state-of-the-art approaches that are addressed in this thesis. We advise that experts in the fields of requirements modeling and product line engineering should only briefly look at these chapters and focus their reading on the main contributions presented in Part II.

Part II Part II introduces the SPREBA (Software Product line Requirements Engineering Based on Aspects [Glinz, 2008b]) approach. First, Chapter 5 presents the basic idea and prerequisites for SPREBA and explains how these prerequisites avoid the problems identified in Chapter 4. Chapter 6 introduces all extensions to the language and visual notation of an integrated modeling approach that are required to realize SPREBA. Chapter 7 further shows how a conventional, static aspect weaving approach (or a static compositional approach in general) must be refined to provide the capability of feature weaving, which is crucial for the stepwise, incremental product derivation realized with SPREBA. Chapter 8 illustrates SPREBA's stepwise, incremental product derivation approach, which combines the product generation and the configuration into a single, seamlessly integrated activity. Chapter 9 presents a more formal semantics of the variability-relevant model elements and how these are translated into an equivalent Boolean formula as a SAT problem. It also presents the detailed algorithms required for this translation process and for the further SAT-based automated analysis, which includes the calculation of constraint propagations and the resolution of any conflicts during a product derivation. Chapter 10 finally introduces feature unweaving, which automates most of the challenging clerical and intellectual tasks required when specifying a variable feature with a compositional approach (e.g., aspect-oriented modeling) in an integrated modeling language. This eases the transition from a requirements reference model to a SPREBA product line model.

Part III The empirical validation of SPREBA is presented in part III of this thesis. First, the tool implementation of the presented concepts with the ADORA tool is presented as a constructive validation in Chapter 11. Further, a general feasibility evaluation of ADORA and SPREBA is presented in Chapter 12, which shows that the approach has already been used successfully for a range of different types of product line exemplars and by different people. In Chapter 13 an empirical performance evaluation of the presented SAT-based automated analysis solution is presented. This evaluation includes the generation of feature diagram-like models, of rather complex SPREBA models and of random generated 3-CNF SAT problems with a critical

clauses-to-variables ratio. It shows that SAT solving the first two types is actually well-behaved and scales only mildly super-linearly, compared to the latter type, which is not well-behaved and scales badly for larger models. Chapter 14 finally presents a real-world case study that systematically evaluates the practical feasibility ADORA and SPREBA in comparison to state-of-the-art UML and feature modeling tools and found some significant benefits of the presented approach.

Part IV

Finally, Part IV presents the conclusions and outlook in Chapter 15. It briefly summarizes the thesis and its contributions, lists all the major limitations of this thesis, and it provides a brief description of the possibly most fruitful future research in this area.

CHAPTER 2

Requirements Modeling

A core activity in requirements engineering is the documentation of requirements. There are numerous languages, templates, and approaches to do so [Glinz, 2006]. However, all of these have particular disadvantages and trade-offs. Today, graphical modeling as an approach to requirements documentation is regarded a promising solution by many authors, e.g., [Glinz, 2006], [van Lamsweerde, 2009], or [Pohl, 2010]. This chapter briefly introduces fundamental concepts in software requirements modeling and briefly surveys the two concrete requirements modeling languages UML and ADORA. These concepts are crucial for a solid understanding of this thesis.

2.1 Fundamentals of Software Requirements Modeling

general model theory

Modeling allows us to plan or reason about changes in reality. When considering any original in the real world, engineers often need to ask questions about operations on this original that can not be executed directly. Depending on the type and nature of the operation this may be too expensive or simply not possible because of physical constraints. By representing all relevant properties of the real world original in a model, though, it is possible to reason about such real world operations of interest based on the model. Stachowiak has described this theory as the general theory of modeling [Stachowiak, 1973, p. 139]. Figure 2.1 illustrates this general model theory as also presented in [Glinz, 2005], which can be applied to any original and operation.

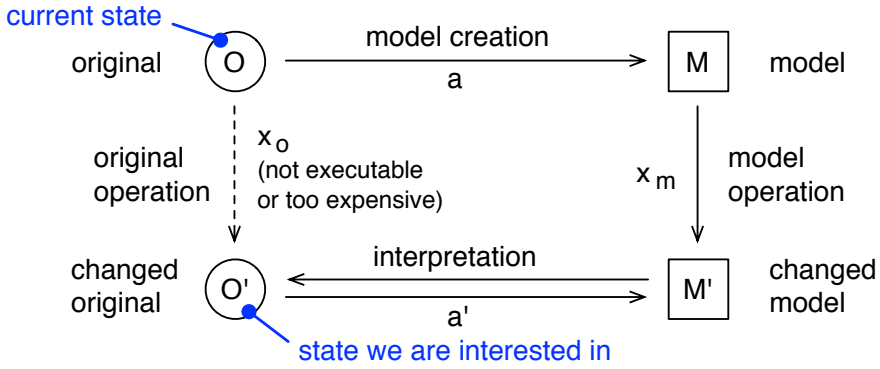


Figure 2.1: Stachowiak's reasoning about original operations via modeling, model operations, and interpretation [Stachowiak, 1973, p. 139] (Figure is drawn as in [Glinz, 2005]).

model definition

Not every entity is a model. Stachowiak has defined the following three criteria that must hold, such that an entity can be called a *model* [Stachowiak, 1973]: First, there must be a natural or artificial original (an object or phenomenon) that is represented by the model (*mapping criterion*). Second, the model represents not all attributes of the original, but only a reduced set of attributes that are relevant for the modeler or model user (*reduction criterion*). And third, models are not per se dedicatedly linked to their originals, but they rather fulfill a purpose for a specific subject (that learns from the model and/or uses the model for any activity), within a specific time interval and constrained by specific mental and actual operations; in short, a model has to be valid and beneficial to use instead of the original (*pragmatic criterion*).

Note that Stachowiak's mapping criterion does not explicitly require the original to actually exist. The original can be an artificial one or another model, too. Thus, it is valid to create a model of an original that is yet merely planned (i.e., does not yet exist), suspected, or that is generally fictitious (i.e., if one would draw a model of how warp drive technology of Star Trek[1] works; then the original would be fictional, i.e., it does not really exist, but the model would be a valid model). Analogously, prescriptive or descriptive models of software are also valid models, despite the fact that software is immaterial and may be planned before it actually exists.

models in software engineering

The scenario shown in Figure 2.1 does to some degree already describe the abstract problem of software requirements engineering. Ludewig surveyed the use of modeling in software engineering and highlights common pitfalls and research issues [Ludewig, 2003]. He also showed that O (Figure 2.1) can be interpreted as the current state (e.g., of a company's business processes and IT), M as a descriptive model of this state, M' as a prescriptive model that may represent optimized processes and IT, and O' as an actual implementation of M' in the real world, which does not yet exist. When planning or reasoning about an operation x_o in an original, this

[1] A popular science fiction entertainment franchise created by Gene Roddenberry.

operation could also be performed on a model M of the original, which leads to the changed model M'. This changed model can then be reviewed with key stakeholders and the quality of the planned operation can be evaluated. Models and model operations can be created at a lower cost than prototypes and experiments with prototypes. Changes in models can be done more easily. Similarly, requirements models are beneficial for the development of software systems as well.

costly requirements errors

As already mentioned earlier, Boehm has shown that the cost of correcting an error increases significantly the later in the software development process it is found [Boehm, 1981]. Errors that are introduced only very late in the development process, e.g., in the testing phase, do not become overly expensive to find and rectify. Errors that were already made in an early requirements engineering phase, however, will most likely be much more expensive to rectify late in a software project. Errors made in a very early phase may even pose a significant threat for the software project to fail as a whole. Therefore, a reasonable requirements specification (a model M of the current state O, the desired development project x_m and a model M' that describes the desired changed original O', as in Figure 2.1) and a reasonable validation of this specification (reviewing M' with all key stakeholders) can potentially avoid misunderstandings and errors during early analysis phases. This can save unnecessary and potentially exorbitant costs of rectifying such errors later-on in the software development process.

complexity

The functionality required of today's software systems is complex and will continue to grow even more complex in the future. Because computers rapidly become faster, with the progression of Moore's law, more and more of the complexity of software-intensive systems is handled by software. Most software systems are further used in socio-technical environments. They need to fit into their environment, provide appropriate interfaces and satisfy crucial non-functional requirements (e.g., quality requirements [Glinz, 2008a]). Requirements modeling may be crucial to some degree, to master this growing software complexity.

separation of concerns

Over the last decades the principle of separation of concerns has been leveraged heavily to cope with the ever growing complexity of software systems. The term has originally been coined by Dijkstra [Dijkstra, 1976] and Parnas [Parnas, 1976], who addressed the problem of separating the code of different features, as e.g. interpreted by [Apel and Kästner, 2009]. Ghezzi et al. provide a more recent and broader definition and regard separation of concerns as dealing with different aspects of a problem individually, in order to master the inherent complexity of software development [Ghezzi et al., 2002]. They state that a concern can be nearly any issue about a software system, ranging from technical (e.g., things like features, functions, reliability, efficiency, environment, user interfaces) to process (e.g., organization, scheduling, design strategies) to economic and financial matters. Ghezzi et al. also note that the separate analysis of different views of a software model, as it has become very popular during recent decades, e.g., within the UML, is also one important type of separation of concerns.

requirements modeling today

Today's state-of-the-art requirements modeling languages follow the principle of separation of concerns by modeling different facets of a software system in different diagrams, see [Pohl, 2010, Chapter 6] or [van Lamsweerde, 2009], for example. The Unified Modeling Language (UML), which has been evolved since the late 1990s, was designed to model the required structure, behavior, user interaction, etc., of a software system in dedicated, separate diagrams. A detailed description follows in Section 2.2. The UML is the de-facto standard software modeling language today. However, there still exist alternative approaches, like ADORA [Glinz et al., 2002], for example, which use only a single, integrated diagram to model all facets of the requirements model. Understanding the basic design of these two approaches to software requirements modeling is important to understand the contribution of this thesis. Therefore, Sections 2.2 and 2.3 illustrate the UML and ADORA in more detail. To prepare this description, however, we first need to briefly recall other general concepts like abstract and concrete syntax, meta-modeling, semantics, and functional requirements.

abstract and concrete syntax

To understand how a modeling language and its notation are built, it is crucial to distinguish between abstract syntax and concrete syntax [Kleppe, 2008]. In natural languages research, Chomsky has originally defined the meaning of abstract syntax to be the "hidden, underlying, unifying structure of a number of sentences" [Chomsky, 1965]. He pointed out that the same fact in an abstract syntax can be defined with different concrete syntaxes (note that today we may actually call these concrete data structures, as concrete syntaxes are more on a language and notational level). For example, the active form "a specialist will examine John" and the passive form "John will be examined by a specialist" [Chomsky, 1965, page 23] differ in their concrete representation (concrete data structure), but both have the same meaning (abstract syntax). In software languages these terms are used similarly [Kleppe, 2008]. By concrete syntax we understand the concrete visual representation of a diagram that is an instance of a given metamodel. The same model can be shown in multiple, different concrete syntaxes—in a textual form or in a graphical form, for example. As abstract syntax we understand the representation of the concepts the language provides, abstracting from the concrete visual representation. To explicitly deal with abstract syntax so-called metamodels are used. A metamodel specifies the available syntax of a given language in a formal way (e.g., with a grammar or a class diagram). These definitions are in-line with [Kleppe, 2008], but differ slightly from Chomsky's because Chomsky also considers the *semantics* as part of the abstract syntax [Chomsky, 1965]. In software modeling the semantics (i.e., the meaning of elements in a given language) is typically dealt with as a separate problem. The semantics of a modeling language is concerned with the meaning of the model elements used in a metamodel. The description and mapping of abstract syntax model elements (e.g., in a meta-model) to the semantic domain, however, is out-of-scope for the contribution of this thesis. See [Harel and Rumpe, 2004], for example, for an introduction into this area.

meta-modeling

The concept of meta-modeling with class diagrams has become very popular in software engineering since the origins of the UML. Before that, grammars, like the extended backus-naur

form (EBNF) [Wirth, 1977], for example, were the predominant notations for describing abstract syntax. Favre argues that what we call a "metamodel" in Modelware corresponds to what is called a "schema" in Documentware or a "grammar" in Grammarware, etc. [Favre, 2004]. In contrast to grammars, metamodeling allows a clearer visualization of the interrelationships between classes of model elements, in the form of a visual diagram. Whether metamodeling should in general be the method of choice for describing abstract syntax is yet questionable, though. For example, Xia has favored the use of grammars for visual specification languages [Xia, 2005] and also Harel and Rumpe mentioned that while class diagrams appear to be more intuitive than graph grammars, they are less expressive [Harel and Rumpe, 2004].

functional requirements

Finally, whenever we model a constructive requirements specification of a planned software system we usually focus on functional requirements. Functional requirements specification, however, does not cover all requirements for a software system. Glinz has pointed out that in every current requirements classification—for example, [IEEE, 1998], [Kotonya and Sommerville, 1998], or [Lamsweerde, 2001]—there is a distinction between requirements concerning the functionality of a system and other requirements [Glinz, 2007]. Today, there is a rather broad consensus about the term functional requirement, which is that a functional requirement is one that requires specific functionality or behavior of a system or product. The nature of non-functional requirements (or quality requirements [Glinz, 2007]) and how to ideally describe them, however, is still hotly debated among requirements engineering researchers. This thesis, hence, focuses on functional requirements modeling only. Non-functional requirements are out of scope, but should still be addressed in future research, in this context.

2.2 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a general-purpose object-oriented modeling language for software requirements and design. The development of object-oriented methods and notations that later evolved into the UML has become popular in the early 1990s. In 1997 the UML was accepted as a standard by the Object Management Group (OMG, a standardization consortium with more than 800 industry member companies). The UML has originally emerged as an industry standard, primarily developed by industry consortia and not within academia. Thus, real-world needs have had precedence over academic rigor. While the UML has become a quite mature language today [Object Management Group, 2010a] [Object Management Group, 2010b], it still suffers from various issues. The following gives a brief overview of the UML's abstract syntax, concrete syntax, visual notation, and issues.

2.2.1 Abstract and Concrete Syntax

abstract syntax

The abstract syntax of the UML is defined in the UML Infrastructure document [Object Management Group, 2010a], which defines the core metamodel, and in the UML Superstructure document [Object Management Group, 2010b], which extends the Infrastructure specification and focuses more on defining the concrete syntax and notations. Essentially, the UML's infrastructure and superstructure specification can be merged into one comprehensive metamodel [Object Management Group, 2010a, page 9], which defines the UML's abstract syntax.

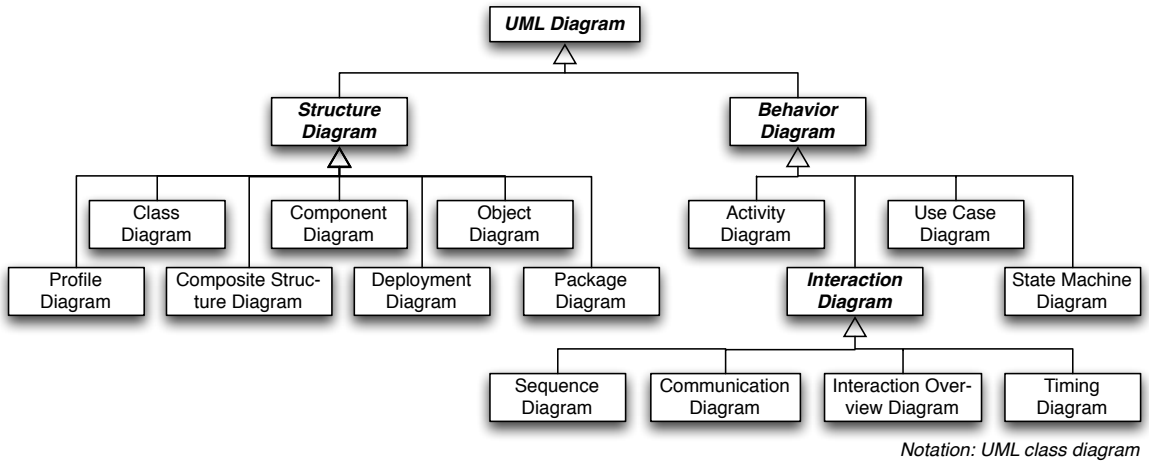


Figure 2.2: An overview of UML 2.3's concrete syntax, which builds on fourteen dedicated, separate types of diagrams [Object Management Group, 2010b].

concrete syntax

Software engineers and analysts who use the UML are typically much more familiar with the UML's concrete syntax. From their point of view the UML is in the first place a set of diagram types that can be used to model various facets of a software system. These diagram types are described in the superstructure document [Object Management Group, 2010b]. This description includes the visual notation and semantics of every of these fourteen diagram types. The UML distinguishes between three classes of diagrams: structure, behavior, and interaction diagrams. Every of these classes consists of several diagram types, which are actually used by engineers and which have a defined visual notation. Figure 2.2 gives an overview of these three classes and their various diagrams types. Each diagram type has its own well-defined syntax and visual notation. One could also create a dedicated metamodel for every UML diagram type. There actually exists tool support to automatically derive a metamodel for a single UML diagram type, which only contains the necessary elements and thus has a considerably reduced size compared to the UML's overall metamodel [Bae and Chae, 2008]. Figure 2.3 illustrates the big picture and shows where the abstract syntax and concrete syntax of the UML can be distinguished.

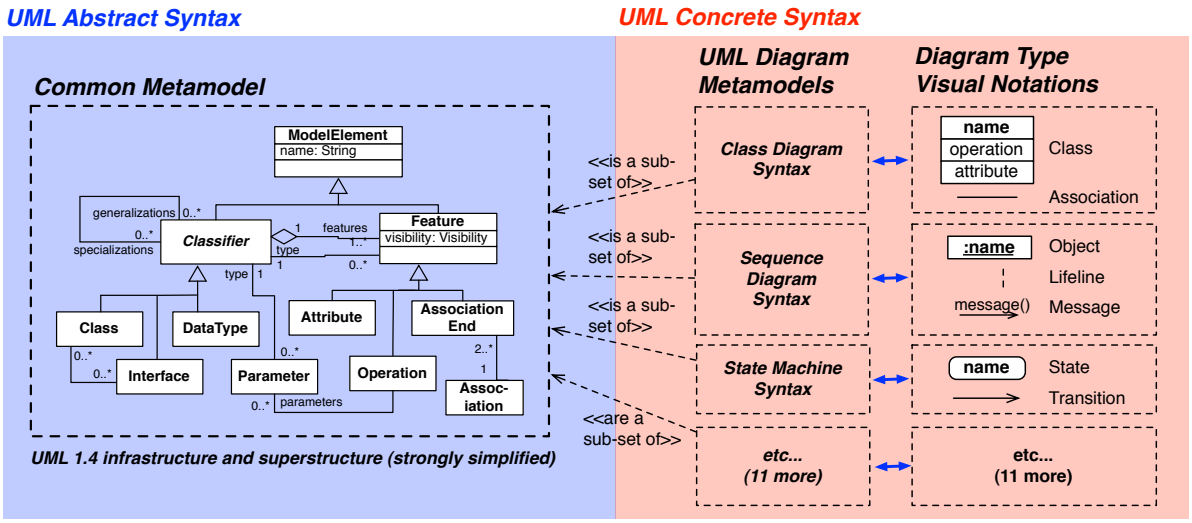


Figure 2.3: An overview of UML's abstract syntax and concrete syntax, where the latter splits the model into multiple separate diagram types and visual notations.

multiple diagram types

As a result of this split of the abstract syntax into several complementary diagram types, the UML offers no possibility to model or edit an instance of the full abstract syntax directly. The only way a UML user can create or edit a UML model is via various concrete diagrams in one of the fourteen different diagram types. This separation can be seen as an advantage because it makes the actual modeling in one diagram at a time simpler and hides much of the abstract metamodel's inherent complexity. However, this separation into multiple diagram types can also be disadvantageous for understanding the model from a user's point of view. Also, when the representation of *features* becomes an important concern, for example, the detailed specification of variable features will inherently be split across all these diagram types. Since the language only offers the fourteen diagram types as shown in Figure 2.2, more abstract concerns like commonality or variability typically impact all of them. We call this problem *information scattering* and address it as a major problem for the contribution of this thesis later-on (Chapter 4).

2.2.2 Other Issues

dealing with inconsistencies

The fact that the UML scatters the concrete syntax of the model over several diagrams of different types (recall Figures 2.2 and 2.3) also leads to new problems of uncertainty and inconsistency between these diagrams. Egyed has presented examples of UML diagrams where the specified information in every diagram was per se correct, but when taken together these diagrams were inconsistent with each other [Egyed, 2006]. In several industrial UML models

Egyed has found numerous such inconsistencies—for example, in the largest case study (that contained 125'978 UML model elements) 1'650 inconsistencies were found. Also Lange and Chaudron reported that despite advances in modern UML tools the number of undetected defects that remain in UML models in practice is alarmingly large [Lange and Chaudron, 2006]. They also find the large number of diagram types that the UML allows to be used with varying degree of rigor a major source of this problem. Their experiments confronted the participants with given fragments of UML models and their results show that most defect types were only detected by less than 50% of the participants. Thus, the UML's scattering of the model into many different diagram types actually leads to a large number of (sometimes serious) defects and misinterpretations. Based on their results Lange and Chaudron raise the hypothesis that domain knowledge may support implicit assumptions that might be wrong and, thus, cause misinterpretations. They suggested guidelines for creating better UML models and for rectifying the most risky defects first in quality assurance.

In further work Egyed has also presented an approach to generate choices to fix inconsistencies that are already successfully discovered [Egyed, 2007]. Moreover, Egyed et al. evolved this concept by removing all false choices that a modeler receives for fixing an inconsistency [Egyed et al., 2008]. Since every choice may affect different consistency rule instances, false choices, for example, are those that again introduce new inconsistencies while fixing one. Tools as in [Egyed et al., 2008] allow to identify and generate reasonable choices for a fix of an inconsistency by checking and evaluating a set of given consistency rules. However, in many cases it may not be possible to detect and/or fix all inconsistencies. Egyed states that modelers have to live with at least some inconsistency, but that even in those cases it is better to be aware of it than otherwise. Egyed et al.'s approach is limited to a syntactical level, however, because the UML does not provide a formal description of its semantics [Harel and Rumpe, 2004].

usage and perceived benefits

Today, there is a large number of open source and commercial tools that support UML modeling. Grossman et al. showed that in the software development community worldwide IBM Rational Rose and Sparx Systems Enterprise Architect are among the most used tools for UML modeling [Grossman et al., 2005]. Their survey also revealed that the use case diagram, the class diagram and the sequence diagram were clearly the most used diagram types (about 90% of UML users use them), followed by the statechart, the activity diagram, the object diagram and others (about 63% and less). On the other hand, the use of the UML to support the software development process is not perceived purely positively. Grossman et al.'s study further revealed that the UML was only latently perceived as efficient and often referred to as *"a technology that is complex, poorly understood, and used inconsistently"*. They state that there is no consensus on how the UML should be used or on whether it is providing beneficial effects, despite its growing adoption throughout the world. However, these numbers and perceptions are likely to have improved in the meanwhile.

ineffective visual notation

The UML standard also comes along with a standardized visual notation for UML diagrams, which is specified in the UML superstructure document [Object Management Group, 2010b].

This description has been based on expert consensus and is almost totally lacking explicitly stated design rationale for its graphical conventions [Moody and Hillegersberg, 2009]. Also, the UML's visual notation is cognitively only little effective [Moody and Hillegersberg, 2009] [Moody, 2009]. In fact, the lack of design rationale for the UML's visual elements leads to many symbol redundancies. For example, rectangles and straight lines are highly overused symbols. Moody argues that each graphical symbol should have a unique value on at least one visual variable (where the visual variables are shape, texture, brightness, size, color, and orientation) [Moody, 2009]. In the UML class diagram, for example, twenty different types of relationships between classes exist and over forty graphical symbols are used in the class diagram notation. Only three visual variables (shape, brightness, and texture) are used to distinguish them, however [Moody, 2009]. Other modeling languages in domains other than software have nearly perfected the cognitive effectiveness of their visual notation. Consider cartography, for example. For software models, however, Moody's systematic analysis has shown that the cognitive effectiveness of the UML's visual notation still performs quite poorly in nearly all regards [Moody, 2009].

mental maps of diagrams

Finally, existing research has shown that keeping the mental map of a diagram is beneficial when relearning a diagram that has been changed [Eades et al., 1991] [Lee et al., 2006]. The mental map is the relative orthogonal ordering, proximity and topology between all model elements. Splitting the UML model into many separate diagrams actually makes the complexity of keeping the mental map significantly simpler and the drawbacks of losing the mental map less disadvantageous because the single diagrams are smaller and less complex by themselves. In most UML modeling tools the mental map is actually kept when a model is saved and re-opened. When a model is edited, it is fully left to the responsibility of the user to change and extend the mental map. When the model is changed or re-generated by a model transformation, however, then the mental map typically gets lost. In such a case the new layout gets automatically re-generated for the new model and this leads to a reduced effectivity when humans need to navigate and search for information in this diagram, which they have previously already seen in a different graphical arrangement. The UML does not foresee concepts like smart layout arrangements or view generation (e.g., where the remaining diagram gets re-arranged in a way that preserves the mental map when edit operations are performed or when parts of the diagram are filtered out)—see [Reinhard, 2010], for example. Modern tools do typically not change the hitherto manually created layout (i.e., mental map), which may results in sub-optimal layouts and cause white space where elements got filtered out. Hence, when model transformations are performed on UML diagrams (for example, when aspects are woven, as follows in Section 2.4), the resulting new diagrams typically are automatically re-layouted as a whole and the mental map of the previous diagrams gets lost.

2.3 The ADORA Approach

ADORA's key concepts

ADORA is an alternative approach to the UML that builds on different assumptions. ADORA stands for Analysis and Description of Requirements and Architecture and is an integrated requirements modeling language and tool. As introduced in [Joos, 2000] and [Glinz et al., 2002] and later re-elaborated in [Meier, 2009] and [Reinhard, 2010], ADORA builds on the following five key principles:

- modeling the structure of the requirements model with *abstract objects* (i.e., prototypical instances of types/classes),
- facilitating *hierarchical decomposition* for hierarchically structuring the requirements modeling (e.g., in the structural, behavioral, and user interaction description),
- building on an *integrated modeling notation* to visualize the model (i.e., structure, behavior, user interaction, and context of the functional requirements model are all presented together in single, integrated visual diagram),
- using *view generation* to create abstract views and to keep potentially large ADORA models visually scalable (i.e., by a combination of fisheye zooming, smart line routing, and automatic label placement for aesthetically appealing layout),
- and allowing users to model different parts of a specification with a *varying degree of formality* (i.e., support for semi-formal requirements modeling, where details can be left out of the model when the importance and risk are low).

2.3.1 Basic Language Concepts

ADORA language

The very first language foundations, that later led to the development of the ADORA language, were introduced in [Glinz, 1995]. Glinz has presented an integrated formal model of scenarios based on statecharts and also a sketch of how this scenario model can be extended into a general system model [Glinz, 1995]. This description already included the idea of modeling the structure of a software system not with classes, but primarily on the level of objects. The ADORA language was first presented and illustrated in [Joos et al., 1997], where it was highlighted that diagrams based on abstract objects allow a superior decomposition of the specification, compared to a structural modeling with class diagrams. ADORA is an integrated graphical modeling language and tool for requirements and high-level architecture modeling. A general description of the ADORA research project can be found on its official webpage[2]. Berner et al. performed an experimental evaluation of ADORA in comparison with the UML, where they found strong evidence that ADORA models are indeed easier to comprehend than UML models [Berner et al.,

[2] See http://www.ifi.uzh.ch/rerg/research/adora.html (checked on June 27, 2012).

1999]. Joos has presented the language concepts of ADORA (i.e., object-orientation, hierarchical and integrated modeling, communication between objects, integration of behavioral modeling, description of object functionality, and modeling of type characteristics), an overview of the ADORA language (i.e., how these language concepts are used) and a comprehensive and reasonably precise language definition that also includes the design rationales [Joos, 2000]. To specify the diverse facets of a system, ADORA uses, for example, a hierarchical model of abstract objects at its core and builds on a notation very similar to Harel's statecharts for behavioral modeling [Harel, 1987], a notation similar to Jackson diagrams for scenario modeling [Glinz et al., 2002], and external actors and external objects to model the system's context.

integrated visualization

Berner et al. also presented a novel visualization concept for ADORA, which is based on the notion of fisheye views [Berner et al., 1998a]. Berner's visualization concept allows displaying local detail and global context in the same diagram and it allows a user to easily navigate in such hierarchical structures while still keeping the overall graphical layout of the model (i.e., the model's mental map). Berner et al. already presented a sketch of the algorithm for the implementation of this kind of 'fisheye' zooming [Berner et al., 1998a]. This work on fisheye zooming in ADORA is similar to Storey and Müller's more widely known and earlier work on the SHriMP (simple hierarchical multi-perspective) visualization technique [Storey and Müller, 1996]. Later-on, Reinhard et al. further presented an improved tool implementation of this original visualization concept [Reinhard et al., 2008].

Overall, a comprehensive presentation of ADORA's basic principles, language definition, visualization concepts, and validation compared to the UML can be found in [Glinz et al., 2002].

ADORA *models and types*

The core building blocks of ADORA models are abstract objects, in contrast to the UML, where these are classes. The ADORA language introduces the concepts of types, abstract objects and concrete instances of objects. Joos has originally explicitly foreseen an orthogonal type directory to specify types and relationships between them [Joos, 2000]. Types were motivated to more efficiently handle recurring specifications in objects that are of the same type and, thus, to reduce redundancy. Abstract objects that are of the same type may share some of their attributes and operations—similar to how this is realized with classes, inheritance, and concrete objects in the UML. In [Joos, 2000] this type directory has been defined as a separate, orthogonal modeling notation. This additional separate type directory could—strictly speaking—be argued to invalidate ADORA's key characteristic of integrated modeling. However, it has been claimed that experience in the ADORA project has shown that the explicit definition of types is only a minor concern when modeling in ADORA. The type directory has also never been implemented with the ADORA tool and has not been addressed in recent work by [Meier, 2009] and [Reinhard, 2010]. The definition and systematic use of types may be important for a precise and efficient ADORA modeling in real-world projects, though. Our recent research indicated that a type-level specification could still be fully integrated into the ADORA approach. This can be done by specifying types as extensions to the ADORA metamodel for every new project. Hence, a single and comprehensive abstract and concrete syntax is still possible in any way, without

splitting the concrete syntax into an ADORA model and a separate type directory. Also, aspect-oriented modeling in the concrete syntax could be considered as an alternative solution as well. Realizing types as extensions to the metamodel (either manually with tool support or also via aspect weaving) seems more natural and feasible to us, though. Such a re-elaboration of the ADORA approach itself, however, is out of scope of this thesis. A sophisticated realization of such an integrated typing concept is subject to future work.

abstract objects

A specification in ADORA is based on a model of abstract objects, where types are only supplementary [Glinz et al., 2002]. Abstract objects are not concrete instances of types, but only prototypical ones. Abstract objects do not have concrete values assigned to their attributes and they are not in any specific state (as real objects at run-time are) [Joos, 2000]. Associations and relationships between concepts are defined on the level of abstract objects. Abstract objects can also be defined as object sets, which can have arbitrary minimum and maximum multiplicities to allow a more detailed specification. Joos has considered types in ADORA as a very intensional concept (i.e., purely on type-level), abstract objects as basically an extensional concept which still contains intensional descriptions like attributes or behavior, and concrete objects (e.g., object instances at run-time) as a very extensional concept and out of scope in ADORA [Joos, 2000]. Abstract objects, hence, can be used on the level of objects but still contain much of the important type-level specification.

2.3.2 Abstract and Concrete Syntax

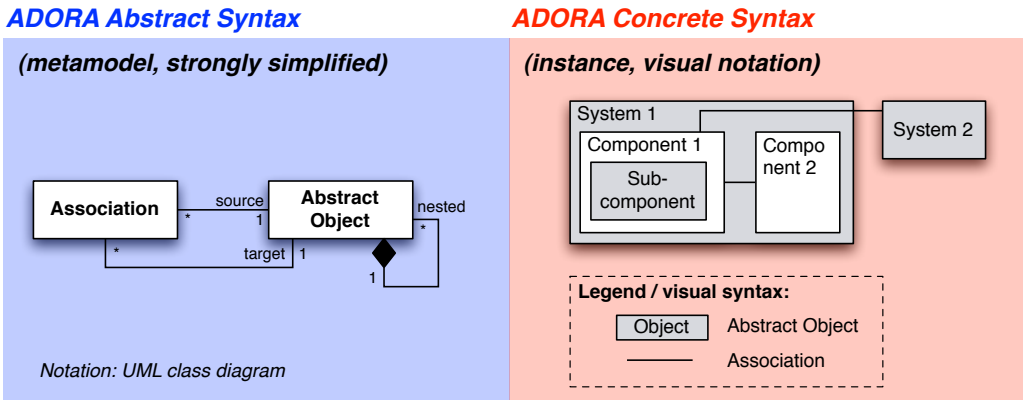


Figure 2.4: An abstract visualization of ADORA's abstract and concrete syntax, covering only abstract objects (i.e., structural modeling).

abstract and concrete syntax

Figure 2.4 shows the abstract syntax of ADORA's base view, which yet consists only of abstract objects and associations, and the concrete syntax, which represents an example instance of the abstract syntax in ADORA's visual notation. This diagram gives a brief overview of how

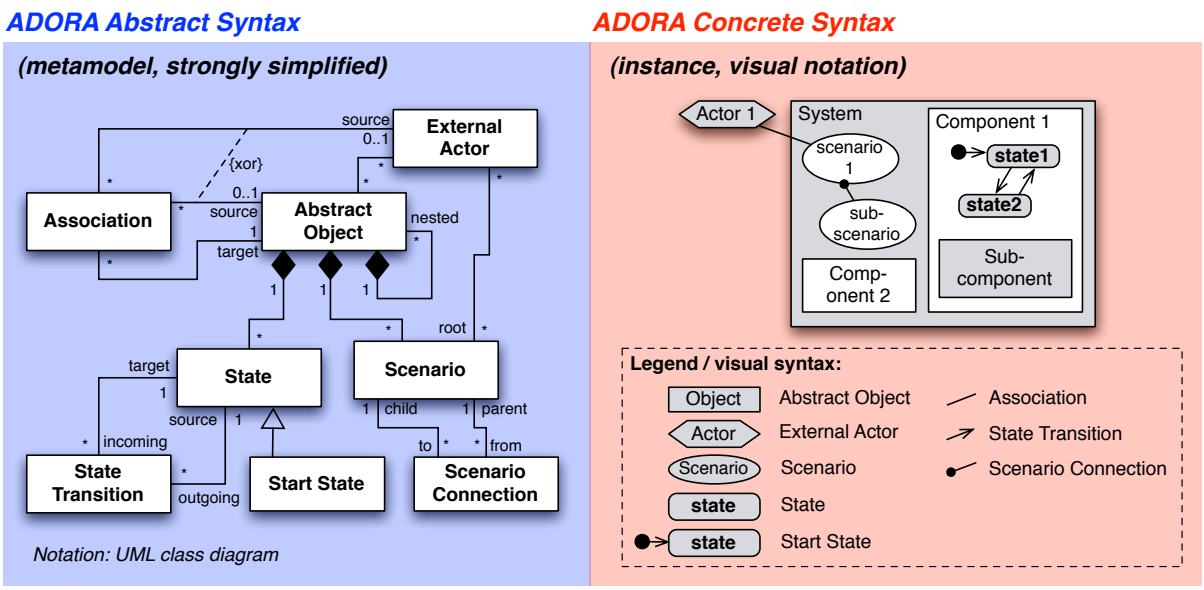


Figure 2.5: An abstract visualization of ADORA's abstract and concrete syntax, covering abstract objects, state charts, and scenario charts (i.e., structural, behavioral, and user interaction modeling).

the model can be visualized with an integrated visual notation. ADORA further integrates all other views into this core metamodel and visual notation as well. These other views include the system's behavior, its context, the use cases (i.e., in the form of type-level scenarios), and more (see [Glinz et al., 2002] for a complete description). Figure 2.5 further also visualizes how these additional views are integrated. The metamodel at the left-hand side of Figure 2.5 shows roughly how system behavior, user interaction, and external actors are dealt with in the ADORA metamodel. Furthermore, the right-hand side of Figure 2.5 shows how such a more comprehensive metamodel can directly be visualized as an instance of the abstract syntax. This model is the actual ADORA diagram, which gets created and edited in a fully integrated visual notation.

integrated concrete syntax

An ADORA model, hence, has a fully integrated concrete syntax and visual notation and does *not* split the abstract syntax into separate, complementary sub-languages as the UML does (recall Figure 2.3). Early empirical work with ADORA has already shown that this integration of all views into one single diagram yields an improved understandability of the model, compared to non-integrated approaches [Berner et al., 1999]. However, such a complete visual integration also leads to new problems of scalability because ADORA models quickly grow large and become difficult to handle. For dealing with these scalability issues a novel concept for view generation has been developed, as follows.

2.3.3 View Generation

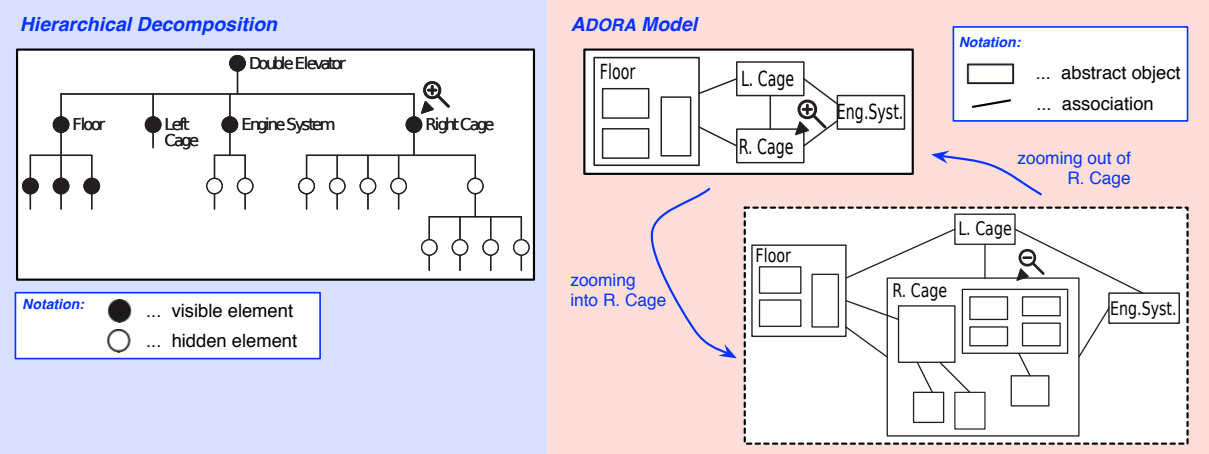


Figure 2.6: The hierarchy and visibility of ADORA objects (left-hand side) and how they are visualized in the ADORA language (right-hand side), as originally presented in [Berner et al., 1998a].

fisheye zooming

ADORA's core model visualization concept, which exploits so-called fisheye views and automatically adapts the layout of the surrounding model elements, has originally been introduced in [Berner et al., 1998b] and [Berner et al., 1998a]. The main component of this original visualization concept was Berner's fisheye zoom for ADORA, which exploits the model's hierarchical structure to create a more abstract view. Figure 2.6 illustrates this. On the left-hand side Figure 2.6 shows the hierarchical decomposition of an example ADORA model, which also shows the visibility of the specified model elements. And on the top-right-hand side it shows the ADORA model in its concrete syntax, which visualizes only the visible objects as shown in the hierarchical decomposition on the left. This ADORA diagram, thus, already presents a dynamically generated abstract view. The components *Engine System* and *Right Cage* are only shown in a collapsed form, which hides all their internal modeling and compresses the overall visual layout. Berner et al.'s fisheye zooming solution allows *zooming into* the object *Right Cage*, for example, which reveals all the previously hidden nested objects, while the surrounding layout gets adapted automatically [Berner et al., 1998a]. This zooming operation is shown in the bottom-right in Figure 2.6. Similarly, such zooming operations can be performed with any hierarchical model element that contains nested model elements in ADORA.

automatic layout adaptions

Berner's fisheye zooming concept presented the idea of seamlessly adapting the level of abstraction of a given ADORA model by expanding or collapsing any node in the decomposition hierarchy at any time during the modeling. Berner et al. also present the basic algorithm to adapt the surrounding layout, as illustrated in Figure 2.7. Whenever a user expands an abstract

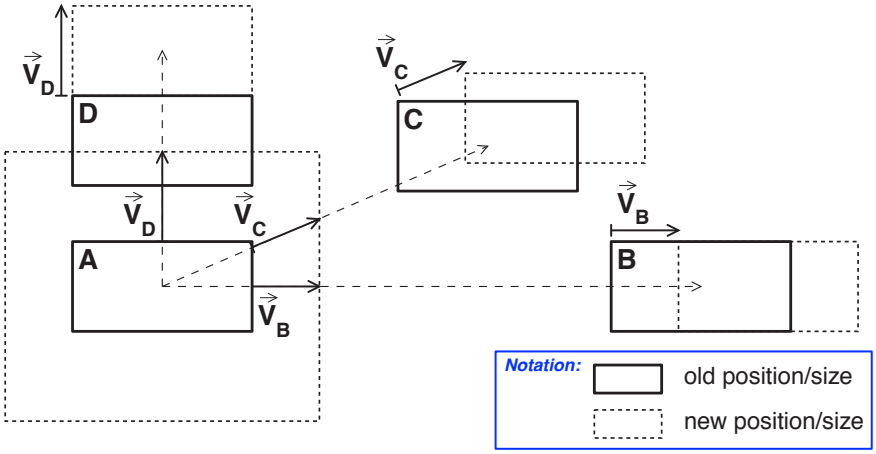


Figure 2.7: ADORA's dynamic and automatic layout adaption concept, as originally presented in [Berner et al., 1998a].

object that is currently collapsed, then all surrounding objects are moved away, such that all the internals have enough space to be fully visualized. Figure 2.7 briefly visualizes how this adaption of the surrounding layout is achieved, where A is a collapsed object that gets expanded: a two-dimensional vector is created from the center of the object that gets expanded to the center of every surrounding object on the same hierarchical level (the level that contains the node on which the zooming operation occurs in Figure 2.6's hierarchy tree). The increased distance between the old boundary of the expanding object and the new boundary will be the distance to which all surrounding elements will be moved apart. If this expansion operation is performed on any node that is itself nested within multiple other nodes in the object hierarchy, then such an operation works recursively, too, by repeating the operation onto all higher levels in the nesting hierarchy. In Berner's doctoral thesis, this fisheye zooming solution is presented in more detail, along with an early approach for line routing that avoids overlaps between lines and other objects in the model [Berner, 2002]. In [Seybold et al., 2003], a tool demonstration of these concepts is presented.

filtering nodes

Seybold et al. also presented how nodes of a specific type can be filtered dynamically in ADORA [Seybold et al., 2003]. Figure 2.8 visualizes this concept based on an example taken from [Reinhard, 2010]. Nodes of different types can be filtered individually in ADORA. As shown in Figure 2.8, for example, all nodes of the type *scenario* can be filtered-out in a specific view and the ADORA tool can generate such a dynamically adapted view that presents only the remaining elements and compresses the surrounding layout at the same time. The basic algorithms are again similar to those shown in Figure 2.7.

improvements and stability

Reinhard et al. later-on developed a new algorithm for line-routing in hierarchical models (e.g., ADORA models) [Reinhard et al., 2006]. This new line-routing algorithm produces an esthetically appealing layout, routes in real-time, and preserves the secondary notation (i.e., the layout

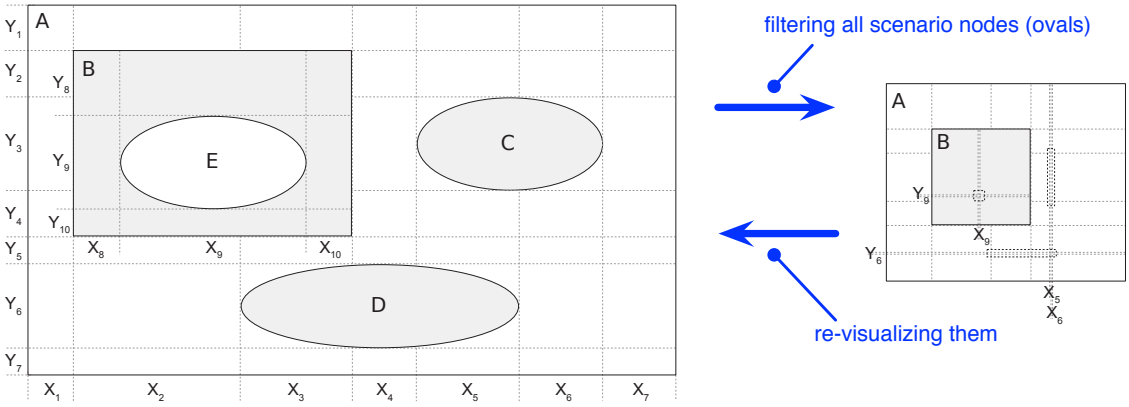


Figure 2.8: An illustration of ADORA's node filtering concept, showing how the layout gets adapted when all nodes of a specific type are filtered or re-visualized, as presented in [Reinhard, 2010].

information) of the diagrams as far as possible. Later-on, Reinhard et al. presented an improved fisheye zooming algorithm that also works flawlessly when the diagram is edited (i.e., nodes or lines are added or changed) between multiple zooming operations [Reinhard et al., 2007]. The solution presented in [Reinhard et al., 2007] also maintains the mental map of the model as far as possible between zooming operations. The presented algorithm allows a user-editable layout that is stable (i.e., no nodes can overlap) also under multiple zooming operations. This solution allows arbitrary zooming operations in ADORA that always preserve the mental map of the model, too. It runs in real-time and was also implemented in the ADORA tool [Reinhard et al., 2008]. As a further aid to improve the visual layout of diagrams Reinhard and Glinz presented a preliminary algorithm for an automatic placement of labels (e.g., event and action descriptions of state transitions) [Reinhard and Glinz, 2010]. This concept can relieve a modeler from tedious manual placements of labels that are annotated to lines in a visual diagram. In his doctoral thesis, Reinhard presented all these further improvements and an experimental validation, which evaluated the benefits of fisheye-zooming in comparison to using only scrolling and explosive zooming (i.e., linear scaling of the model) in ADORA [Reinhard, 2010].

2.3.4 Other Issues

dealing with inconsistency

Inconsistency between diagrams of different types, as in the UML (recall Figure 2.3), can not occur such easily in ADORA, since ADORA's concrete syntax and visual notation are fully integrated. ADORA allows the definition of precise rules for consistency between different facets of the model [Glinz et al., 2002]. An integrated modeling language like ADORA also contains less redundancy than a non-integrated one like the UML, which leads to fewer constraints that

are required to enforce consistency between the different facets [Joos, 2000]. There also exists a comprehensive grammar for the ADORA language as a whole [Meier, 2009]. ADORA has formally defined language constraints, which are predicates that define how an instance of a model element is connected meaningfully with another model element of the language [Xia, 2005, pp 35]. Language constraints in ADORA are strictly enforced. This means that they are checked immediately, before a model edit operation gets executed [Meier, 2009]. Invalid ADORA models, thus, can not be created in the ADORA tool.
Whether it is beneficial to restrict the editor to a degree that allows only the creation of correct models is debatable, however. The more restrictive a modeling editor is, the more inflexible the editor may occur to users and this may decrease the overall usability. However, the benefits are that all created models will be valid, which avoids much otherwise laborious work on manually evolving the model into a correct state. Xia and Seybold presented a large set of language constraints for ADORA, see [Xia, 2005] and [Seybold, 2006]. And in [Cramer, 2007] a further and more up-to-date discussion of these language constraints is presented. While the current version of ADORA implements mechanisms that check all these defined constraints, this realization could still be improved to gain a better runtime performance. For example, a general purpose logic programming language like Prolog could be used in future work to verify the conformance of the model against the ADORA grammar more efficiently. This would be straightforward, as ADORA's grammar is already formally defined in [Meier, 2009].

simulation

ADORA models can also be simulated interactively. Seybold has presented an approach and tool to do so [Seybold, 2006]. Seybold's approach even allows evolution of semi-formal ADORA models into fully formal ones. His approach processes all the user interaction steps in a semi-formal ADORA model and prompts the engineer to enter new information to complement the model whenever there is some information missing in the behavioral specification. This allows engineers to validate ADORA models by simulation and to evolve semi-formal ADORA models into more complete and consistent models through simulation. While simulation is not implemented in the current version of the ADORA tool, the necessary concepts exist. An implementation can help to evolve a semi-formal ADORA model into a fully formal one and can also help to validate and verify the specification together with customer stakeholders.

tool implementation

All the above mentioned capabilities of ADORA have been implemented in a tool as a validation of their constructive validity, see e.g. [Seybold et al., 2003], [Seybold, 2006] or [Reinhard et al., 2008]. The latest version of the ADORA tool is available under an open source license [RERG, University of Zurich, 2011]. This current version does not implement ADORA's simulation capabilities, since Seybold chose to validate his concepts on basis of a previous tool implementation that was still a pure Java application [Seybold, 2006]. ADORA is implemented as a set of plug-ins for the Eclipse IDE and uses the Eclipse graphical editing framework (GEF) as a basis for its visual representation. The ADORA tool implementation has always been an evolutionary prototype and was intended to foster explorative research and to constructively validate ADORA's conceptual research. It has not been developed as an industrial-strength tool. It does

provide a solid and comprehensive implementation, nevertheless. In [Meier, 2009, chapter 11] a detailed description of ADORA's tool implementation is provided. Chapter 11 provides further insight on the ADORA tool, but focuses on the contributions of this thesis.

weak empirical validation

From a scientific point of view a major concern about ADORA is its weak empirical validation. Besides the work presented in this thesis the only real validation of the ADORA approach against the state-of-the-art (e.g., the UML) has been performed in [Berner et al., 1999]. This study has presented a controlled experiment with fifteen advanced students of informatics and systematically evaluated the empirical benefits when comprehending a given ADORA model in comparison to an equivalent set of UML diagrams. The results were clearly in favor for ADORA [Berner et al., 1999]. The study participants found that visualizing the model in an integrated way significantly improves the understandability, compared to visualizing it with multiple separate diagrams. However, both the ADORA and the UML have significantly evolved since 1999. UML 2.0 has been introduced and the most recent version of the UML to date is 2.4.1. Many problems that were present in earlier versions of the UML, as presented in [Glinz, 2000], for example, have been mitigated or solved in the meanwhile. Later major research efforts on ADORA have only validated their new ideas against conventional ADORA, e.g. [Xia, 2005], [Seybold, 2006], [Meier, 2009] and [Reinhard, 2010]. Therefore, from a scientific point of view, it is today in fact not known or evaluated whether visualizing real-world software models with ADORA provides a better understandability than visualizing them with UML. Furthermore, there is no empirical evidence whether the ADORA approach together with its advanced visualization techniques, as presented in [Reinhard, 2010], can really provide a similar scalability, understandability and usability as state-of-the-art UML tools can. Empirical data in this regard does not yet exist for ADORA. Further, Berner et al. also did not consider any state-of-the-art tools in their evaluation [Berner et al., 1999].

no industry adoption

Despite the availability of the ADORA tool implementation under a commercial license [RERG, University of Zurich, 2011], ADORA has not yet been adopted by companies in industry. Also, lessons learned from real-world projects do not yet exist, therefore (i.e., for conventional ADORA and without considering the work presented in Part III of this thesis). The ADORA tool, hence, might still have yet unnoticed children's illnesses from an industry user's point of view, which may not yet be discovered because of this absence of adoption.

ineffective visual notation

ADORA's visual notation is quite similar to the one used in the UML. Objects, associations, states, and state transitions, for example, are denoted equally. Scenarios are denoted as ovals, similar to UML use cases. In general, as it is the case for the visual notation of the UML [Moody, 2009], also ADORA's visual notation was not designed with a clear rationale to optimize the visual notation for a good cognitive effectiveness (e.g., as proposed in [Moody and Hillegersberg, 2009]). Similarly to the UML's situation, an improved visual notation that is optimized for cognitive effectiveness that reflects the modeling language's syntax and semantics well is still a subject for future research.

limited expressibility

There are still other facets of a software system that cannot be modeled appropriately in the ADORA language. From a modeling perspective these include data flow (e.g., as with data flow diagrams) or business processes (e.g., as with process chain modeling or activity diagrams), for example. Also, not all information that can be specified in ADORA gets graphically visualized in the ADORA tool, today. For example, attributes and operations of abstract objects can be specified in the functional view in a textual form, see [Meier, 2009], but they can not be visualized graphically in ADORA's current visual notation, as they can be visualized in UML class diagrams, for example.
When taking a wider view on early textual requirements and on source code, test cases, etc., this information can also not straightforwardly be described in a well-defined way in ADORA (i.e., other than a mere annotation of textual comments and/or traceability links). However, this is a heavily researched area in the context of model-driven engineering (MDE) around the UML as well. In ADORA, an integration with early phase textual requirements has already been addressed by Habr, for example, who used a mapping-based approach [Habr, 2008]. To date, the visual expressibility of ADORA is still limited to the modeling of structure, behavior, user interaction and context of a software system, which embraces a narrower scope than recent specifications of the UML, for example. This thesis extends this scope to some extent and also includes variability and variability constraints modeling, as follows in Part II.

orthographic modeling

Atkinson et al. addressed this expressibility problem for fully integrated visual notations quite comprehensively and came up with the idea of an *orthographic modeling environment* [Atkinson and Stoll, 2008] [Atkinson et al., 2010]. They propose using a so-called *single, underlying model* that includes *all* views of a software system, including: goals, early-phase requirements, models, code, tests, etc. While this idea sounds attractive, the approach is still lacking a working tool prototype. We think that the work on ADORA goes into a similar direction. ADORA provides an integrated concrete syntax and visual notation that can specify and visualize the requirements specification and high-level architecture in a single model. Views are then generated to allow visualizing particular facets of the model only. Source code, for example, could be integrated in a way that is similar to how the functional specification of objects is handled in ADORA [Meier, 2009]. However, today, ADORA's expressibility is still limited. Limitless expressibility as suggested with Atkinson et al.'s orthographic modeling would be very desirable for ADORA as well. However, this clearly is future research.

2.4 Aspect-Oriented Modeling (AOM)

tyranny of the dominant decomposition

Tarr et al. describe the general problem that aspect-oriented modeling (AOM) deals with [Tarr et al., 1999]:

All modern software formalisms support separation of concerns to some extent, through mechanisms for decomposition and composition. However, existing formalisms at all lifecycle

phases provide only small, restricted sets of decomposition and composition mechanisms, and these typically support only a single, "dominant" dimension of separation at a time. We call this "tyranny of the dominant decomposition".

...leads to cross-cutting concerns

In the majority of today's modeling approaches the dominant mechanism of decomposition is object-orientation. In the UML this manifests primarily in a decomposition and description of the system with components and classes and in ADORA this manifests in a hierarchical decomposition with abstract objects. This universal type of decomposition allows a developer to perform an adequate separation of concerns of the software system's structure. However, other facets of the software system now cannot be completely freely decomposed anymore to reach an ideal separation of concerns there, too. This is because they need to conform and match to the object-oriented (i.e., structural) decomposition already. For example, in the UML, the software behavior is typically described by defining state machines for specific classes or components. In ADORA, the statechart-based specification of the system's behavior, for example, also has to be aligned with the general object-oriented structure. Therefore, Tarr et al. argue that the modular structure of an artifact achieves separation of concerns only along the dominant dimension [Tarr et al., 1999]. Hence, for other facets of a software system, like behavior or user interaction, for example, an adequate separation of concerns often cannot be established. This leads to so-called *cross-cutting concerns*, which cause scattering and tangling. Such cross-cutting concerns impede a clean and complete separation of concerns.

aspect-oriented modeling

To mitigate these problematic issues of scattering and tangling of functional cross-cutting concerns, *aspect-orientation* as been introduced. In object-oriented modeling, a system is primarily decomposed into separate (highly cohesive and lowly coupled) modules by the object-orientation paradigm (e.g., with classes, components and/or abstract objects). Cross-cutting concerns, however, can not be decomposed into dedicated objects, since their functionality needs to be realized among several other objects that have previously been formed through the object-orientation paradigm. The "tyranny of the dominant decomposition", hence, hinders any clean modularization of all given concerns [Tarr et al., 1999]. Aspect-orientation aims at breaking this tyranny, by introducing what one may call a second-tier dominant decomposition of the system into core concerns (i.e., normal modeling) and aspectual concerns (i.e., aspect-oriented modeling). Such aspect-oriented modeling allows a clean modularization of homogeneous cross-cutting concerns (i.e., concerns that require a specific piece of functionality equally in multiple other concerns). Hence, AOM allows to eliminate all redundancy that would otherwise be unavoidable. An aspect introduces a dedicated model for a specific cross-cutting concern and untangles all other concerns from the specification of the cross-cutting concern. This improves the overall separation of concerns. However, aspect-orientation also comes with disadvantages. For example, AOM introduces new problems for the understandability of a model, it may break the principle of information hiding and the join points of aspects may be fragile. In this regard, Meier has already presented a detailed criticism on aspect-oriented software development and has also provided a detailed comparison of nearly all state-of-the-art

approaches for aspect-oriented software development all over the software life-cycle [Meier, 2009, pp. 23-52 and appendix A].

focus on functional modeling

In this thesis, we take a much narrower scope, which only addresses functional software requirements modeling. Some authors in the area of aspect-oriented requirements engineering consider not only functional cross-cutting concerns, but also non-functional requirements as candidates for modeling with aspect-oriented approaches [Chitchyan et al., 2005] (non-functional requirements describe qualities of functional requirements, like e.g. performance requirements). However, today, there is still no consensus in the requirements engineering community about what non-functional requirements actually are and how they should be dealt with [Glinz, 2007]. Hence, non-functional requirements can hardly be modeled properly with state-of-the-art languages like the UML or ADORA, as already discussed in Section 2.1. The discussion of AOM in this thesis, thus, focuses on functional requirements modeling.

terminology

The most basic terms one needs to know about aspect-oriented modeling are the following. An *aspect* is made of an *advice* and a *pointcut*. The *advice* is the actual functionality that is cross-cutting (e.g., the model elements that would otherwise need to appear redundantly at multiple locations). The *pointcut* specifies precisely where and how the advice needs to be *woven* into the remaining functionality. *Weaving* is the process of composing an aspect at all coordinates listed in the pointcut. These coordinates, where an advice needs to be woven in, are called *join points*. For every given modeling language a precisely defined *weaving semantics* needs to exist, such that the definition of an aspect is precise and unambiguous and such that tools can automate the weaving process. An *aspect* includes all model elements that constitute a cross-cutting concern, which are the advice and the pointcut (i.e., the protocol, with which these model elements need to be composed).

The term *aspect*, in general, has ambivalent meanings, though. In a much wider context, for example, the term aspect has also been used to denote different views on a software system, like, e.g., the behavioral or user interaction aspects (i.e., views) of a system. These are sometimes called aspect-views, or similar. When considering how the term aspect is defined in common dictionaries it turns out that it is quite nearby to denote such aspect-views as aspects, too. Hence, one could possible invent a better term for modularized cross-cutting concern than the already overloaded term *aspect*. However, the aspect-oriented programming (AOP) and aspect-oriented software development (AOSD) communities consistently use the term *aspect* to denote modularized homogeneous cross-cutting concerns. Thus, the term *aspect* is also used analogously in this thesis. Therefore, whenever we refer to an “aspect-view” in a wider context, we use the term *facet*, in this thesis.

2.4.1 AOM with UML

There are numerous approaches to aspect-oriented modeling. Chitchyan et al. already presented a very comprehensive survey on aspect-oriented analysis and design approaches [Chitchyan et al., 2005]. Because of space constraints, we will therefore only introduce one of the most advanced solutions by Whittle et al., see [Whittle and Jayaraman, 2007] and [Whittle et al., 2009], which roughly partitions the state of the art into symmetric and asymmetric approaches and explicitly targets the UML.

symmetric vs. asymmetric

Whittle et al. argue that, broadly speaking, there are two kinds of approaches for modeling aspects in the UML [Whittle et al., 2009]. The first approach is the *symmetric* one. In this approach two models are composed by identifying common model elements and merging these. Such common elements may for example be two classes with the same name. This approach is implemented in Theme/UML [Clarke and Baniassad, 2005] and also in the work of France et al. [France et al., 2004]. The second approach is the *asymmetric* one. Whittle et al. argue that the asymmetric approach essentially reuses the join point model from aspect-oriented programming [Kiczales et al., 1997] at the modeling level, for specifying and weaving aspects. Further, they argue that a significant amount of research, including the work presented in [Cottenier, 2006] and [Jacobson and Ng, 2004], has basically identified a join point model for a modeling language and then simply used AspectJ advices [Kiczales et al., 2001] of before, after, and around for the weaving.

MATA: superior expressiveness

MATA (which stands for Modeling Aspects Using a Transformation Approach) is more expressive than existing symmetric or asymmetric approaches because any model element can be a join point or an advice [Whittle et al., 2009]. This allows a nearly completely arbitrary decomposition of a model into separate aspect models. Based on an empirical evaluation with students Whittle et al. show that existing symmetric and asymmetric solutions fail to specify some of the required composition strategies for state diagrams [Whittle et al., 2009], when applied to realize the use case slice technique presented in [Jacobson and Ng, 2004]. Jacobson and Ng's use case slices aim at splitting the model into fragments that exactly match a use case and compose these only late in the development process. Whittle et al.'s results show that both symmetric and asymmetric approaches have trouble in adequately expressing some required composition strategies, or just can not express them. Neither of them were expressive enough to handle all model compositions and join points that were relevant. A simple merging of states as in existing symmetric approaches cannot be used for a more complex weaving of system behaviors, which requires a weaving of actions and transitions in a very specific way. Only relying on a subset of the base language's model elements to be used, as join points as in the existing asymmetric approaches, seems to be overly restrictive to the authors and cannot realize particular composition strategies. Whittle et al. argue that MATA supports all composition categories because the entire state machine diagram syntax is available for defining a composition.

MATA: concrete syntax-based

Most of the existing symmetric and asymmetric aspect-oriented modeling approaches are based on the abstract syntax of the UML. They are of limited expressibility, however. The MATA approach instead views an aspect composition as a special case of a model transformation [Whittle et al., 2009]. The key difference in MATA is that there are no explicit join points. Instead, graph transformations specify and compose aspects and any model element can be a join point. More precisely, graph rules are used to compose a base model and an aspect model. Building on graph transformations also allows the use of existing well-understood formal techniques. For example, critical pair analysis is used in MATA as an automatic and lightweight method for finding structural interactions between aspect models.
Whittle et al. highlight that MATA graph rules are defined over the *concrete syntax* of the modeling language, in contrast to almost all other approaches to model transformation, which typically define transformations at the meta-level, i.e., the abstract syntax [Whittle et al., 2009]. They argue that this restriction to concrete syntax is important for aspect modeling because a modeler is unlikely to have the detailed knowledge of the UML metamodel to specify transformations over abstract syntax. This is certainly true when considering how the UML was perceived and used until recently [Grossman et al., 2005]. Defining these graph rules in the concrete syntax allows a graphical specification, which is similar to classic modeling and thus probably more reasonable for engineers.

MATA: information scattering

On the one hand, it needs to be possible to specify aspects in the concrete syntax in a successful AOM approach, in order to make the technology accessible to software engineers. On the other hand, modeling aspects on the UML's concrete syntax inevitably leads to information scattering. This happens because multiple diagrams of diverse types are used (recall Section 2.2.1) and a particular aspectual concern needs to be specified with a specific aspect model for every of these diagrams. Thus, multiple aspect models are necessary to specify a single aspectual concern. This still yields an open problem. For example, Jayaraman et al. used MATA to model features as aspects to realize a software product line [Jayaraman et al., 2007]. They represented each feature as a model slice and as an increment over other features. When visualizing these models as proper UML diagrams, though, this still lead to many separate diagrams for representing a feature. Therefore, their aspectual feature specification is still scattered over multiple diagram fragments of different types. Further, when considering the join points as relevant for visualizing the aspect as a whole, too, then the specification may be regarded even more scattered. Information scattering is a general concern with aspect-oriented modeling based on the UML's concrete syntax, though. The reason is that the UML model itself is already inherently scattered over various diagram types. This is essentially a result of the UML's fundamental design, though, recall Section 2.2.

2.4.2 AOM in ADORA

handling cross-cutting concerns

Cross-cutting concerns also appear in ADORA models, as a result of ADORA's object-oriented dominant decomposition. To alleviate this problem, aspect-oriented modeling has been introduced for ADORA, as a second-tier dominant decomposition. Because an ADORA model is primarily decomposed by abstract objects (called the base view or structural view), cross-cutting concerns primarily occur in the system behavior and user interaction views (i.e., the statecharts and scenario charts). Meier et al. therefore introduced a solution for aspect-oriented modeling and aspect weaving in ADORA, see [Meier et al., 2006] and [Meier et al., 2007]. The main aim of this solution was the efficient modeling and evolution of such cross-cutting concerns.

ADORA aspects

Since ADORA is an integrated modeling language, which differs considerably from the UML and other modeling languages, it also required a new solution for aspect-oriented modeling. ADORA's aspect-oriented modeling solution is an *asymmetric* one, similar to the join point model we know from aspect-oriented programming [Kiczales et al., 1997]. However, ADORA's aspect modeling concept is also a visual concept and differs from other AOM approaches as follows. ADORA uses *aspect containers*, which are similar to abstract objects but are not part of the actual model's structure. An aspect container contains all cross-cutting model elements for a particular cross-cutting concern. To define how the aspect needs to be composed, ADORA uses so-called *join relationships*. A join relationship defines the join point (i.e., the location where the advice applies) by the target element and the weaving semantics by the source element in the aspect container and the selected weaving type. The source model element, the target model element and the weaving type of a join relationship define the weaving semantics, hence. An aspect can have arbitrarily many join relationships, but must have at least one—otherwise it does not have any impact on the model. ADORA uses three different weaving types: *before*, *after*, and *instead*. While the first two are similar to the more widely known weaving semantics of AspectJ [Kiczales et al., 2001], the third one also allows replacing other elements and, thus, enables a higher expressibility. These weaving types formally define how the weaving needs to be executed [Meier, 2009].

weaving semantics

Figure 2.9 shows some simple aspect-oriented ADORA diagrams and illustrates how ADORA's aspect weaver composes these aspects for every of the three weaving types, as specified in [Meier et al., 2007] and [Meier, 2009]. The top part of the Figure 2.9 illustrates this for behavior chunks (i.e., a behavior chunk is a behavioral description that is not self-contained but only fragmentary) and scenario chunks (i.e., fragmentary scenario descriptions). For a formal elaboration of any special cases—e.g., when multiple aspects compete at the same join point—please refer to [Meier, 2009]. Meier also shows a more detailed example of a library system with an authentication aspect [Meier, 2009, pp. 107-114]. Figure 2.9 only illustrates a basic overview of aspect-oriented modeling (left-hand side) and how these models are composed into semantically equivalent non-aspect-oriented ADORA models with ADORA's aspect weaver (right-hand side). These examples give a very good overview on ADORA's aspect weaving se-

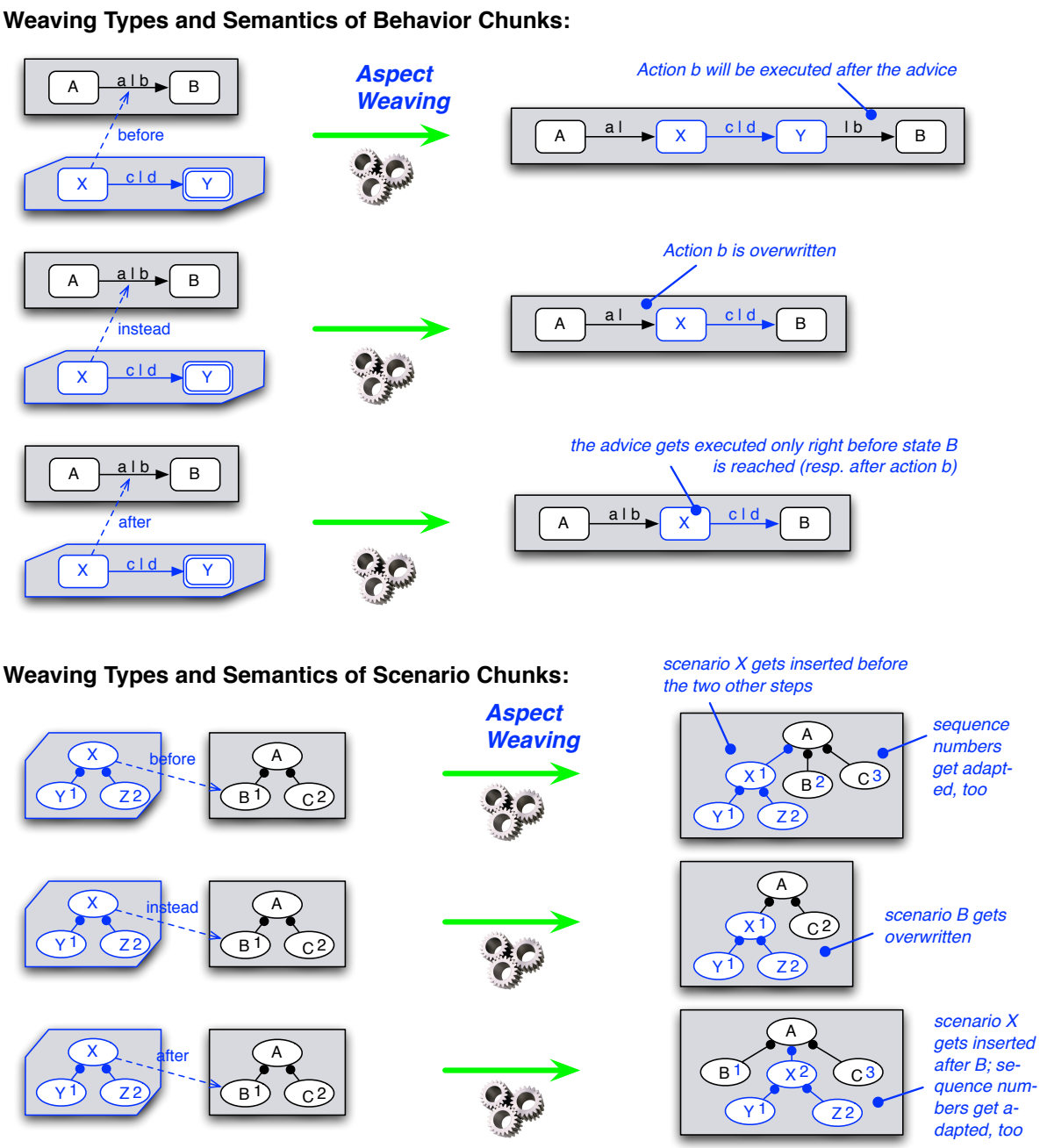


Figure 2.9: An overview of the weaving semantics of behavior and scenario chunks for the weaving types *before*, *instead*, and *after*, as defined by [Meier et al., 2007].

mantics. These weaving operations are implemented and automated in the ADORA tool, see [Meier, 2009, Chapter 11].

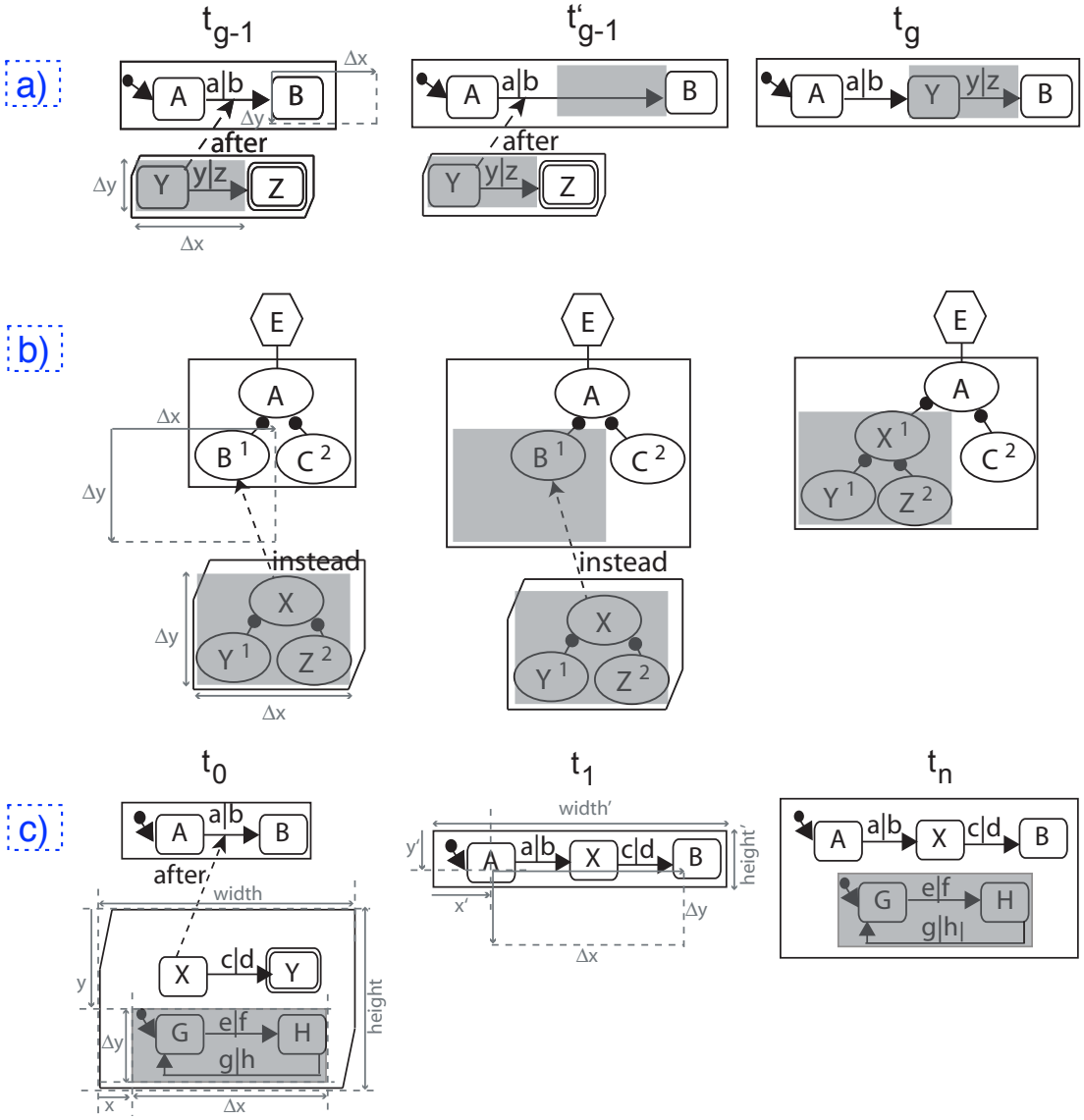


Figure 2.10: An illustration of how the graphical layout is composed when a) a behavior chunk, b) a scenario chunk, and c) a behavior chunk along with other embedded components (in this case with a statechart) gets woven, as defined by [Meier, 2009].

weaving layout information

When a user is manually adding a new model element anywhere in an ADORA model, the precise graphical coordinates where this model element shall be located in the graphical model is always given by the user. Reinhard's layout adaptation techniques, hence, have all the needed input to optimize the layout [Reinhard, 2010], recall Section 2.3.3. When an ADORA aspect gets woven fully automatically by the ADORA tool, however, only the source and target model

elements are known and it remains unspecified at which graphical locations the user may desire the composed model elements to be located, after the weaving. Thus, the tool has to find a suitable position for every newly woven model element automatically. Meier argued that also the layout information of how the model elements were previously arranged within the aspect container should be kept as far as possible in the resulting woven model [Meier, 2009].

For also weaving an aspect's layout information Meier has presented what he called a rudimentary, straightforward solution for weaving the aspectual layout information [Meier, 2009, pp. 202-205]. Figure 2.10 illustrates this solution. Meier's aspectual layout handling approach fully keeps the layout of the advice by including the scenario or behavior chunk at the graphical location of the join point (i.e., the target location of the join relationship) as a whole. This solution also reuses Reinhard's layouting algorithm for model edit operations, see [Reinhard et al., 2007]. Figure 2.10 particularly illustrates how the existing aspectual layout gets composed in a way that preserves the overall mental map of the diagram as far as possible. This solution produces reasonably good layouts for very simple aspects—see Figure 2.10, for example. For more realistic aspects, which may contain several behavior chunks, scenario chunks, and other components, this layouting solution often results in very suboptimal layouts. Kandrical has aimed at improving the graphical weaving of aspects in ADORA [Kandrical, 2009], but could not come up with a solution that performed better than Meier's preliminary one [Meier, 2009, pp. 202-205]. Finding an ideal solution for such a weaver is in general a quite hard optimization problem and is to the best of our knowledge still an unsolved problem. For this thesis, finding an optimal solution for ideally arranging woven graphical layouts had to be out of scope, unfortunately. Preliminary ideas were reflected in [Kandrical, 2009], though. Therefore, finding an overall solid solution to graphical aspect weaving in ADORA is still subject to future research, see Section 15.3.

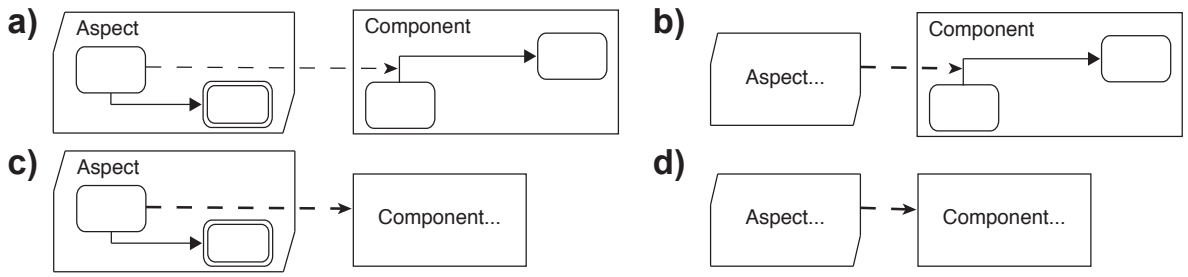


Figure 2.11: An overview of how ADORA creates abstract visualizations of aspect containers and join relationships when parts the model are not visible in a generated view, as defined by [Meier et al., 2006].

views on aspects

Finally, ADORA's aspect modeling capabilities can also visualize aspects at any arbitrary level of abstraction. Arbitrary abstract views, as they can be generated for any plain ADORA model,

recall Section 2.3.3, can also be generated on any aspect-oriented ADORA model. This requires that aspect containers and join relationships can be visualized in an abstract form, too, when some of their crucial details are not visualized in a generated view. For example, when the actual target node of a join relationship is not visualized in an abstract view, then also the join relationship can not be visualized accurately and will be shown as an abstract join relationship [Meier, 2009]. This abstraction mechanism, hence, works very similarly also for aspect containers, as it does with plain abstract objects—recall Section 2.3.3. For join relationships the mechanism is similar to abstract views on associations between objects. Figure 2.11 shows an overview of how aspects in ADORA are consistently visualized on different levels of abstraction. In diagram $a)$ Figure 2.11 shows a simple example aspect-oriented model. In the diagrams $b)$–$d)$ it further shows three situations where abstraction is involved and an abstract representation of the aspect-oriented modeling is necessary. These view generation mechanisms on aspects are fully implemented in the ADORA tool, today. A more detailed specification, well-formedness rules and more can be found in [Meier et al., 2006] and [Meier, 2009].

CHAPTER 3

Variability Modeling

This chapter introduces and motivates variability modeling, with a focus on the concepts relevant for requirements modeling. Today's requirements modeling languages do not inherently support the modeling of software product lines, which requires explicit support for specifying variability and variability dependencies. Neither the UML nor ADORA (i.e., before this thesis) provide inherent support for feature modeling or variability modeling, without extensions. In general, there are various orthogonal variability modeling languages, which can straightforwardly be applied with the UML, as this chapter introduces. Furthermore, the object management group (OMG) is currently also assessing the common variability language (CVL) as a new, standardized variability modeling notation [Haugen and et al., 2010] [Czarnecki et al., 2012]. CVL, however, will also not inherently be integrated into requirements modeling, but is an additional notation that gets mapped to meta object facility (MOF)-based languages as well [Czarnecki et al., 2012]. CVL is not discussed in detail in this chapter, primarily for space reasons. Otherwise, this chapter gives a broad and brief overview on the state of the art of variability modeling today.

overview

Section 3.1 first motivates why it is important to model variability in disciplines like software product line engineering (SPLE). Further, this chapter is structured as shown in Figure 3.1. Section 3.2 generally introduces variability modeling. Section 3.2.1 first introduces major variability modeling languages and notations. Section 3.2.2 further shows how these variability modeling languages are integrated with existing requirements modeling languages and notations. Furthermore, Section 3.3 summarizes the state of the art of automated analysis of variability

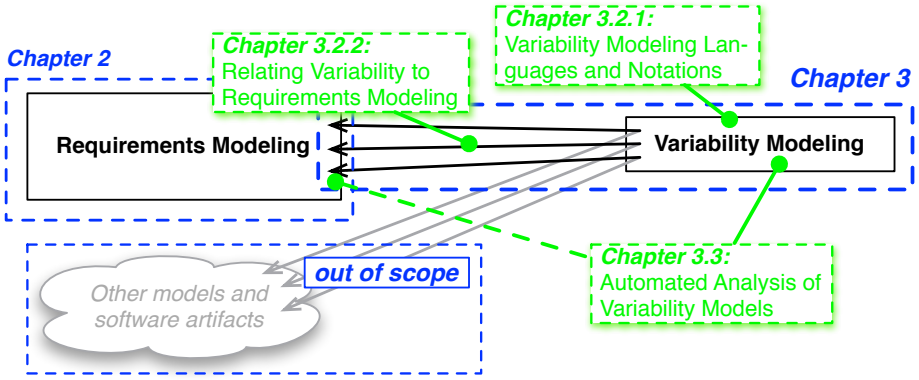


Figure 3.1: An overview of the scope and structure of Chapter 3.

models, which is mostly focused on variability models only (Section 3.3.1), but also includes work that enhances model checking and similar approaches with variability (Subsection 3.3.2). The state of the art in all of these topics needs to be elaborated because the approach presented in Part II of this thesis will introduce both a novel variability modeling language and approach and a novel automated constraint analysis solution for this new language. Experts in the field may skip this chapter and rather focus on Chapter 4 and Part II.

3.1 Software Product Lines and Feature-Orientation

The software industry is different from classic mass production industries, like automobile manufacturing, for example, where product line engineering has originally become popular. Software is immaterial and can be duplicated with almost no material effort. Thus, the problem with software is development, not production. However, the principles of re-using common parts for various products and systematically managing the variability also apply to software development.

history Previous work on software product lines (SPL), or respectively on program families, as it was called earlier, dates back to the 1970ies [Dijkstra, 1972] [Parnas, 1976]. Later-on, in the early 1980s, domain engineering has been developed [Neighbors, 1980], followed by the yet more systematic approaches of feature-oriented domain analysis (FODA) [Kang et al., 1990] and the Synthesis method [Software Productivity Consortium Services Corporation, 1993] in the early 1990s. The first systematic approaches to software product line engineering came up in the late 1990s and the early 2000s, as described in [Brownsword and Clements, 1996], [Weiss and Lai, 1999], [Clements and Northrop, 2001], [Atkinson, 2002], [Gomaa, 2005], or [Linden et al., 2007], for example. Today, the scope of variability modeling also seems to broaden further,

as new research fields like feature-oriented software development (FOSD) [Apel and Kästner, 2009], for example, emerge.
All these concepts are similar to some extent: they all aim at re-using the commonality between sets of closely related software products and at systematically handling the variability. Software product line engineering (SPLE) [Clements and Northrop, 2001] [Pohl et al., 2005] is one of the most systematic approaches to do so. Thus, we will use the term SPLE from here on.

benefits

Hitherto SPLE research was strongly motivated by the potentially huge benefits in development efficiency. The major benefits named in the literature are: (i) significant reduction of development costs for individual products, (ii) faster product development cycles and time-to-market, and (iii) an increase in software quality. Clements and Northrop list even more benefits like productivity gains in general, increased market presence, unprecedented growth, improved customer satisfaction, higher reuse goals, mass customization, and compensation for an inability to hire software engineers [Clements and Northrop, 2001]. The reason for the software quality improvements of SPLE is that all reused software artifacts already went through testing, debugging, and possibly maintenance and evolution phases before and, thus, are actually more likely to have fewer known and yet unknown defects.

fundamentals and definitions

SPLE allows maximizing the reuse of commonality (i.e., by developing all products on a common product platform) and of variability (i.e., by a more modular development of variable functionality that can be added to or removed from the product more easily). This requires a *variability model*, though, to support an efficient specification and development of both the software product line as a whole and of individual application products.

Clements and Northrop define a software product line (SPL) as *"a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of assets in a prescribed way"* [Clements and Northrop, 2001]. Their work, however, focuses little on variability modeling (e.g., feature modeling [Kang et al., 1990]) and more on management, social and organizational aspects that are decisive for success when introducing software product lines in companies.

Czarnecki and Eisenecker provide a much more technical insight into variability modeling and automated product generation, with a focus on programming [Czarnecki and Eisenecker, 2000]. They define feature modeling (as originally introduced in [Kang et al., 1990]) in a quite detailed manner and argue that feature modeling is the key technique for identifying and capturing variability. Further, they highlight that a feature model is an *intensional* definition of a variability model. The set of all allowed application products' feature models (i.e., where all variability is bound and all constraints are satisfied) is the *extension*, then. Czarnecki and Eisenecker also introduce a range of concepts that allow an automated generation of application programs from existing product line artifacts.

Pohl et al. further define SPLE as *"a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization"*, where they define a platform as *"any base of technologies on which other technologies or processes are built"* and mass customization as *"the large scale production of goods tailored to individual customers' needs"* [Pohl et al., 2005]. They introduce a comprehensive approach to SPLE that uses not feature modeling but orthogonal variability modeling (OVM) as the key concept for variability modeling. In [Bühne et al., 2004] the authors have argued that feature modeling is not adequate for modeling requirements variability and leads to several problems. For example, the reasons of variabilities are lost (i.e., OVM introduces the variation point concept to document these) and a local change of a feature may lead to inconsistencies between separate feature diagrams (i.e., where they suggest to use only one common feature model that is also mapped to the OVM). The approach presented in [Pohl et al., 2005] puts a stronger focus on notations and is an intuitive one that addresses the topic in a manner that is similar to how software engineering is taught at universities today.

feature-oriented software development

More recently *feature-oriented software development* (FOSD) has been introduced [Apel and Kästner, 2009] as another research area that focuses on the development of product *features*. Apel and Kästner define the term feature as a *"unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option"*. They further describe the basic idea of FOSD to decompose a software system in terms of the features it provides. They state *"the set of software systems generated from a set of features is also called a software product line"*. Compared to Czarnecki and Eisenecker's definition of intensional and extensional variability [Czarnecki and Eisenecker, 2000], this statement refers to FOSD as an intensional approach, whereas they refer to classic SPLE approaches, as e.g. described in [Clements and Northrop, 2001], as extensional [Apel and Kästner, 2009]. Most SPLE approaches, like e.g. [Czarnecki and Eisenecker, 2000], [Pohl et al., 2005] or SPREBA, as follows in Part II of this thesis, are actually intensional, though.

features as first-class entities

A key argument for FOSD is the use of *"features as first-class entities to analyze, design, implement, customize, debug, or evolve a software system"* [Apel and Kästner, 2009]. Existing approaches to requirements modeling typically consider classes or components as first-class entities, but not features. They are tailored to classic single system development. Prehofer has first introduced features as such a first-class entity into the programming language Java, by exploiting the mechanisms of inheritance and aggregation [Prehofer, 1997]. He called this approach *feature-oriented programming*. Doing so releases a designer from structuring his program primarily into classes, but also allows her/him to define features as primary program artifacts. This leads to a much higher modularity and flexibility. It can also leverage the problem of feature interactions and improve the reusability, since features are artifacts more likely to change from a requirements perspective than other artifacts like, e.g., packages or classes. Batory et al. furthermore developed the AHEAD tool suite, which provides a language-independent model of features and allows the implementation of features in different languages, like, e.g., programs

and grammar specifications [Batory et al., 2004]. FEATUREHOUSE [Apel et al., 2009], further, is another extension into this direction and provides support to enable an easier implementation of the AHEAD concepts for new languages. Ultimately, the FOSD community[1] as a whole is motivated to find a unified theory of FOSD that may realize FOSD's general vision, namely *"the generation of efficient and correct software on the basis of a set of feature artifacts and a user's feature selection"* [Apel and Kästner, 2009].

3.2 Variability Modeling

Variability modeling is said to be *the* key ingredient to realize a software product line. This section gives an overview on variability modeling.

classification

Sinnema and Deelstra have already presented a classification of variability modeling techniques and gave a broad overview on existing variability modeling languages and tools [Sinnema and Deelstra, 2007]. They compare six of them in more detail: VSL, ConIPF, COVAMOF, cardinality-based feature modeling (CBFM) [Czarnecki et al., 2005b], Koalish and Pure::Variants (see [Sinnema and Deelstra, 2007] for detailed references). These languages all have a slightly different focus, ranging from more problem space-oriented variability modeling (e.g., CBFM) to more solution space-oriented ones that already focus on software architectures (e.g., Koalish [Asikainen et al., 2004b]). All of these six languages use a Boolean notion for variable features. Hence, they all consider variable entities (i.e., variable features) as something that is either selected or deselected for a product (this is different in certain decision modeling approaches). The variability modeling languages then provide a syntax and notation to describe these variable entities and their relationships (i.e., hierarchies and other constraints). Sinnema and Deelstra's comparison reveals many subtle differences between these languages and tools. Especially the visual notation for variability differs considerably, ranging from purely textual to purely graphical notations that use quite different visual symbols.

literature survey

Chen et al. further presented the chronological background and summarized key issues that drove the evolution of 34 different approaches to variability management from 1990 to 2008, which they identified in a systematic literature review [Chen et al., 2009]. They found, for example, that only a few of these approaches tackle systematic process support for variability management, that only three of them are concerned with evolution of variability, that only two of them mentioned the scalability as an issue to be addressed and that only one among them explicitly mentions testing. In their conclusion they also state that there is a vital need of conducting a comparative analysis of these various approaches. They state that there is yet only little—if any—experimental or detailed comparative analysis of variability management approaches.

[1] See http://www.dagstuhl.de/11021/ (checked on June 27, 2012) for a recent major meeting.

tools Lisboa et al. furthermore presented a systematic review of existing tools for domain analysis [Lisboa et al., 2010]. Their research questions focused on finding a specific or generic process that these tools may follow, on the tools' main functionalities and on their development and usage. They selected 19 tools for domain analysis and 20 major functionalities (e.g., pre-analysis documentation, matrix of the domain, etc.) to support planning, modeling, validation, and other tasks in domain analysis. One of their findings was that only four of the tools were developed and used purely in practice (five of the academic tools were also used in industry and one of the industry-developed tools was used in academia). Eventually, they conclude that a useful domain analysis tool must support the whole process and not only individual sub-processes. They claim that gaps in functionality make it difficult for industry to adopt a specific tool. Thus, they advise that all these functionalities should be covered within one domain analysis tool (some of these would be quite simple to add). However, specific designs of languages and notations may also have a significant impact on the feasibility of certain functionalities in concrete tool implementations. Languages and notations, techniques and automated analysis solutions, as described in the following, were not covered in Lisboa et al.'s survey.

3.2.1 Variability Modeling Languages and Notations

illustrating four major notations In the following subsections four major variability modeling languages and their notations will briefly be introduced. These should give a good overview of how variability modeling "looks like" today. The covered languages and notations are *AND/OR tables*, *feature modeling*, *orthogonal variability modeling (OVM)*, and *decision modeling*, as follows.

AND/OR Tables

an ad-hoc approach AND/OR tables are a straightforward, ad-hoc approach for variability modeling. This approach is widely used in industry in Switzerland—also in major companies [Fricker and Stoiber, 2008]. In these companies this variability modeling approach has been called 'AND/OR tables', while in other contexts other names have been used for the same notation, like, e.g., 'product matrix' or similar. The approach was also used in literature, see [Muthig et al., 2004], for example. The clear benefit of using AND/OR tables for variability modeling is that it is easy to learn, easy to understand and requires little training and additional costs for software tools. While the approach initially scales very well for supporting managers and engineers during early product line planning and development phases, it inevitably leads to inconsistency problems and various kinds of efficiency issues in the long run. The reason for that is that dependencies between variable entities can not be expressed. Hence, consistency maintenance becomes very laborious.

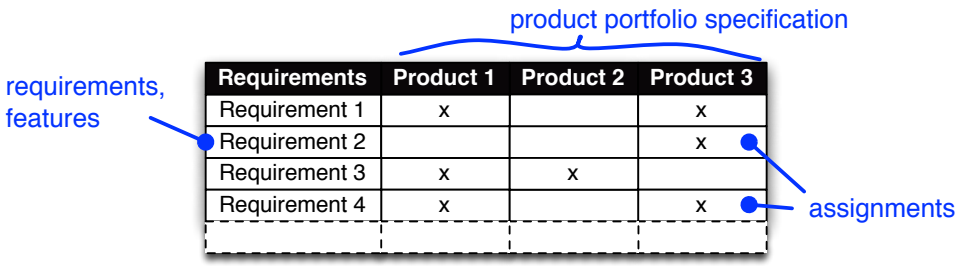


Figure 3.2: Extensional specification of a product portfolio and variability model by using so-called AND/OR tables (an ad-hoc notation which we have frequently found in industry).

Figure 3.2 shows a conceptual illustration of how the AND/OR table approach can be used for the specification of requirements or features (depending on the chosen level of abstraction) and product configurations. Simply checking a requirement in the column of a specific product indicates that this requirement is part of this product's specification. Not checking a requirement for a specific product means that it is not realized, respectively. An AND/OR table essentially reflects the specification of the whole product portfolio (columns) and the list of implemented requirements/features for every of these products (rows, where the boxes are checked).

This approach to variability modeling is purely *extensional*, while nearly all other variability modeling approaches are *intensional*. A major disadvantage is that no hierarchies between requirements/features and no variability constraints (e.g., alternatives, requires dependencies, etc.) can be specified. This, in turn, leads to extra manual efforts when consistency of existing configurations needs to be assured and when the product line must be changed or evolved. The advantage, however, is that AND/OR tables are easy to use, easy to learn, and that the engineer can directly specify the "end-product" (i.e., the product portfolio specification of the planned product line).

Feature Modeling

FODA

Feature modeling was originally introduced by Kyo Kang and colleagues as part of an approach that was called feature-oriented domain analysis (FODA) [Kang et al., 1990]. FODA already aimed at *"the identification of prominent or distinctive features of software systems in a domain"* and at defining *"both common aspects of the domain as well as differences between related systems in the domain"*. FODA allows defining features as mandatory, optional, or alternative product characteristics for a specific application domain. Kang et al.'s original paper has become very well known within the past ten years. Feature modeling today is the most widely known and used modeling notation for describing the commonality and variability of a software product line.

many feature modeling dialects

Since Kang et al.'s original feature modeling notation many variations, dialects, and extensions to feature modeling have been developed. Schobbens et al. have surveyed seven major dialects of feature modeling and highlighted their characteristics in language elements and visual notations [Schobbens et al., 2006]. They also derive a formal semantics of feature diagrams, based on a mathematical notation (in set theory), which reduces ambiguities and makes the notation safer and more efficient to use with tool support.

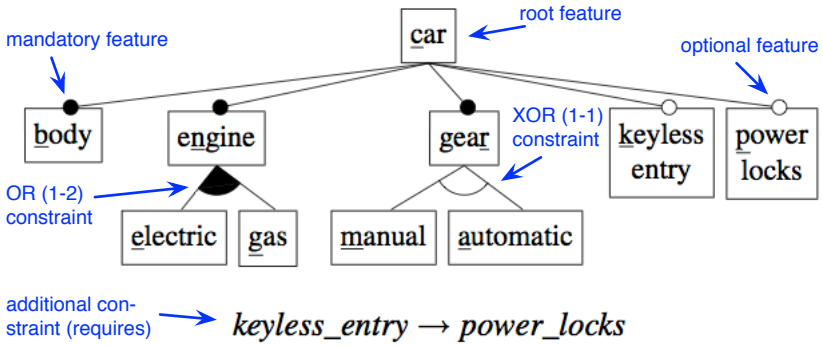


Figure 3.3: A simple feature diagram example of a car, taken from [Czarnecki and Wasowski, 2007].

example and illustration

In Figure 3.3 an example feature model is shown, taken from [Czarnecki and Wasowski, 2007]. Its syntactical language elements are highlighted in blue. Features are typically modeled as rectangles and always have a parent feature, except for the root feature that is on the top of the tree. Whenever its parent feature is chosen, every feature can either be a mandatory feature (indicated with a filled black circle on top of the feature) or an optional feature (indicated with an empty circle). Examples are the mandatory feature $body$ and the optional feature $power\ locks$, see Figure 3.3. Further, there can be cardinality or feature-group constraints defined for multiple features that have the same parent feature. An example for a $1{:}2$ cardinality (i.e., a logical OR) constraint is shown among the two sub-features of the feature $engine$ (as a filled circle segment between the feature decomposition edges). Another example for a $1{:}1$ cardinality (i.e., a logical XOR) constraint is shown among the two sub-features of the feature $gear$ (as an empty circle segment). Note that in Czarnecki and Wasowski's notation features that are part of a group constraint are neither marked as mandatory nor as optional, as all other solitary features are [Czarnecki and Wasowski, 2007]. Finally, additional constraints can be defined in Boolean logic. In Figure 3.3 the constraint $keyless\_entry \rightarrow power\_locks$ has been defined as a simple *requires* dependency.

Orthogonal Variability Modeling

separating variability and concept modeling

Feature models mix two conceptual descriptions in one diagram: the feature model and the variability model. Since the level of features is generally considered as a good level of abstraction for variability modeling, most authors today still rely on feature modeling as their variability modeling language of choice. From a practical standpoint, a feature in a feature model is often just a property of interest and not necessarily a user-visible functional characteristic, however. Hence, it may be argued that orthogonal variability modeling (OVM) and feature modeling are fairly similar notations. Pohl et al.'s work, nevertheless, has closely investigated the fundamentals of variability modeling and came up with the OVM notation [Pohl et al., 2005].

An orthogonal variability model is a model that defines the variability of a software product line and relates it to other software development models, such as feature models, use case models, design models, component models, and test models [Pohl et al., 2005]. The basic elements of an OVM are the following four, which relate to specific questions, as defined in [Metzger and Pohl, 2007]: *Variation point ("what does vary?")*, which documents a variable item or a variable property of an item. *Variant ("how does it vary?")*, which describes the possible instances of a variation point. *Variability constraints*, which are constraint dependencies about certain design decisions or technical dependencies. And *visibility of variability ("who is it documented for?")*, which discerns between internal and external variability, where only external variability is visible to the customers. The core idea of OVM is that it is always orthogonal and mapped to any other notation used during the SPLE process. Figure 3.6 demonstrates this later-on.

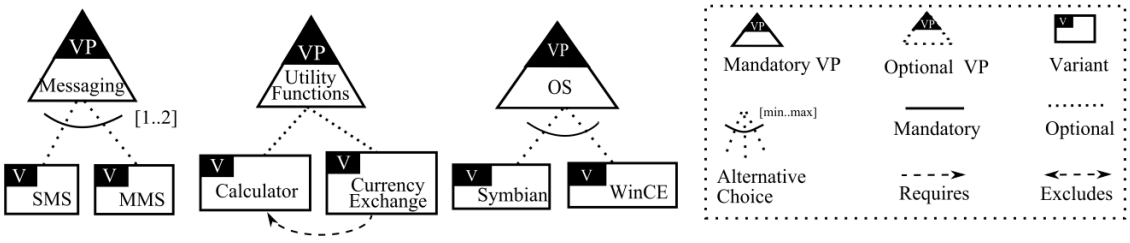


Figure 3.4: A simple OVM example of a mobile phones product line, taken from [Roos-frantz et al., 2009].

example and illustration

Figure 3.4 shows a simple OVM example. It consists of three variation points and six variants on the left hand side and an overview of the OVM notation, as introduced in [Pohl et al., 2005], on the right hand side. The OS variation point consists of two variants, where an alternative choice with default cardinality (i.e., 1:1) is specified. For the $Messaging$ variation point a logical OR constraint is defined among the two variants and for the $UtilityFunctions$ variation point one of the two variants also $requires$ the other one to be selected.

Decision Modeling

There also are various decision modeling approaches like Synthesis, Schmid & John, DOPLER, Vmanage or KobrA, for example, which differ significantly from each other, even in their core concepts, as surveyed in [Schmid et al., 2011]. The decision modeling approach that is the closest to requirements modeling is the one by Schmid & John. For brevity, thus, we will focus on this approach. Schmid et al. [Schmid et al., 2011] later argued that their notation, see [Schmid and John, 2004], builds on the Synthesis approach, see [Kasunic, 1992]. The customizable decision modeling approach of Schmid & John has also strongly influenced and inspired our original table-based Boolean decision modeling concept presented in [Stoiber et al., 2007]. In Part II, however, a much refined and novel decision modeling concept beyond far beyond this initial work will be presented.

the role of Synthesis

In [Schmid et al., 2011] the authors argue that *"similar to the role of the FODA report [Kang et al., 1990] in the context of feature-based variability management most (if not all) existing decision modeling approaches have been influenced by the Synthesis method [Software Productivity Consortium Services Corporation, 1993] which introduced the basic idea of decision models in 1993."* Synthesis, thus, is briefly described as follows. The Synthesis method was introduced in [Campbell et al., 1990], where the concepts *domain engineering* and *application engineering* were emphasized. This report was filed in June 1990, thus five months earlier than [Kang et al., 1990]'s FODA report, which derives its definition of domain analysis from many other, previous references, like, e.g., [Barstow, 1985], [Prieto-Diaz, 1988], [Arango, 1988] or [Batory et al., 1989]. How a decision model shall be used for domain engineering was yet not part of [Campbell et al., 1990]'s report, though. Schmid et al. pointed out that the concept of *decision modeling* is explicitly mentioned and explained in [Software Productivity Consortium Services Corporation, 1993] for the first time [Schmid et al., 2011]. The description of the decision modeling concept in [Software Productivity Consortium Services Corporation, 1993], however, is very informal and mostly based on an example (i.e., the decision model can be represented either as a list of questions or in a tabular format [Schmid et al., 2011]). The general idea of domain analysis dates back to [Neighbors, 1980], though.

example and illustration

The example in Figure 3.5 shows an example decision model in Schmid & John's notation, taken from [Schmid and John, 2004]. It lists three decision items, as shown in the column $Name$. The first decision item $Memory$ is further specified to be relevant only when $System\_Mem = True$. Every decision has a dedicated column for a textual description and a $Range$, where the latter also implies the data types and defines the values out of which the engineers have to take their actual choice. Further, a $Selection$ column defines how often this decision needs to be instantiated and a $Constraints$ column can specify additional restrictions with relational expressions that involve other decision variables. Finally, $BindingTimes$ are defined for every decision to describe when (i.e., at what point in time during the product development life-cycle) this decision can be taken. In order to take the variability binding decisions in this example

Name	Relevance	Description	Range	Selection	Constraints	Binding Times
Memory	System_Mem = True	Does the system have memory?	TRUE, FALSE	1		Compile Time
Memory_Size		The amount of memory the system has (KB)	0..100.000	1	Memory=TRUE => Memory_Size > 0	Installation, System Initialisation
Time_Measure-ment		How is time measurement done?	Hardware, Software	1		Compile Time

Figure 3.5: A simple decision model example of a configurable embedded system, taken from [Schmid and John, 2004].

one could decide $Memory$ as $TRUE$, $Memory\_Size$ as $65'536$ and $Time\_Measurement$ as $Hardware$, for example.

data types vs. only Boolean

Schmid and John introduce five so-called selector types for modeling a decision: optionality (Boolean), alternative (xor among two possibilities), set alternative (xor among many possibilities), set selection (several choices can be made within a certain range) and value reference (a specific value can directly be given to the decision) [Schmid and John, 2004]. This approach of using different data types and ranges for different decisions is unlike *feature modeling* and *OVM*, for example, where all features, respectively variation points and variants, are typically of the type Boolean—i.e., they can only either be deselected or selected as part of the product. While there are exceptions (e.g., [Kang et al., 1990]'s numerical feature *Horsepower*, which also involved numerical constraints) Boolean features are the prevalent case. Allowing arbitrary values makes the configuration space a lot larger. For example, for the decision item $Memory\_Size$ there are $100'000$ choices for the memory size, as defined in Figure 3.5. Whether such a large configuration space is generally feasible and beneficial may be questionable, though. A feature modeling notation would restrict this decision item to a group feature that allows a limited number of choices, e.g., 16k, 32k or 64k byte of memory. These choices would possibly suffice for all stakeholders and simplify the model's complexity considerably (i.e., with regards to verification and automated analysis). Recent work in [Passos et al., 2011] has shown that about 50% of the features in a real-world variability model (i.e., the real time embedded operating system eCos) were indeed non-Boolean, however. But whether Boolean variability suffices also depends on the use of the model. Models closer to implementation or in technical domains such as systems software often contain non-Boolean variables (e.g., as strings that gets spliced into the code, or similar). On the level of requirements such non-Boolean variable features are rarely really necessary, though. We generally advise to favor a purely Boolean variability modeling to keep the complexity low, which makes the model's automated analysis much more scalable and feasible.

3.2.2 Relating Variability to Requirements Modeling

limited expressibility of variability models

All variability modeling languages and notations, as introduced in Section 3.2.1, are of limited expressibility and can not—or only to a very limited extent—specify software functionality or requirements. Feature models, for example, are often described as a "requirements modeling notation", but provide only a very limited notation for functional requirements modeling. They are primarily used for variability modeling. OVM models, on the other hand, use a dedicated notation that deliberately only specifies the variation points and variants. In fact, Pohl et al. explicitly name a list of shortcomings of integrated variability modeling, where the variability is mixed within traditional software development models [Pohl et al., 2005, pp 74-75]. These shortcomings include (i) consistency problems, (ii) comprehension of traceability of variability over the software life cycle, (iii) overloading of single modeling notations, (iv) differences in how variability is modeled in different notations that may not integrate well into an overall picture of variability, and (v) ambiguous definitions when defining variability only within a single development model (e.g., when variability is only defined in a feature model). Therefore, the OVM notation was developed and custom-designed to be used over the whole product development life-cycle as an orthogonal notation. This requires extensive tracing to all other artifacts where variability occurs. Similarly, also decision modeling notations rely on such traceability or variability mappings, respectively.

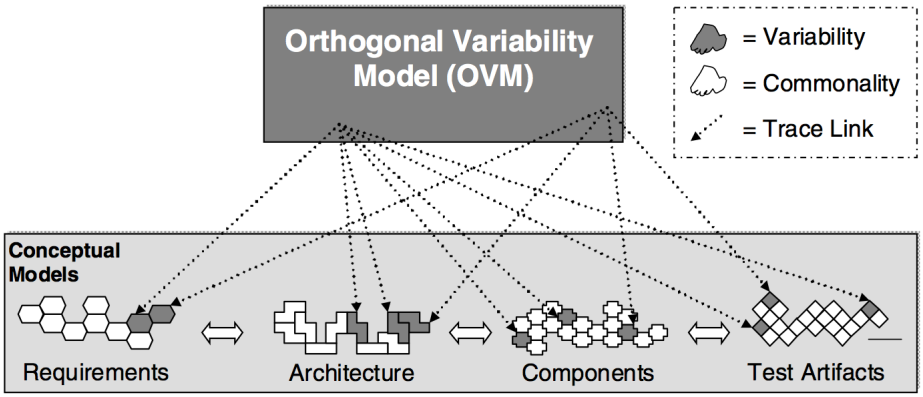


Figure 3.6: An OVM model and its orthogonal relationship to other conceptual models all over the software life-cycle, taken from [Metzger and Pohl, 2007].

variability mappings

The typical approach to realizing a software product line is mapping single variable entities of a variability model to all artifacts in the software engineering life-cycle that should be customizable (e.g., textual requirements, requirements and design models, source code, tests, etc.). In the following the focus is laid on requirements (and design) models. Figure 3.6 shows an abstract view of how an OVM variability model is mapped to different conceptual models of

different software lifecycle phases [Metzger and Pohl, 2007]. The OVM particularly builds on comprehensive reference models and *negative variability* [Pohl et al., 2005], as illustrated in Figure 3.6. There are other variability realization mechanisms as well, however, as follows.

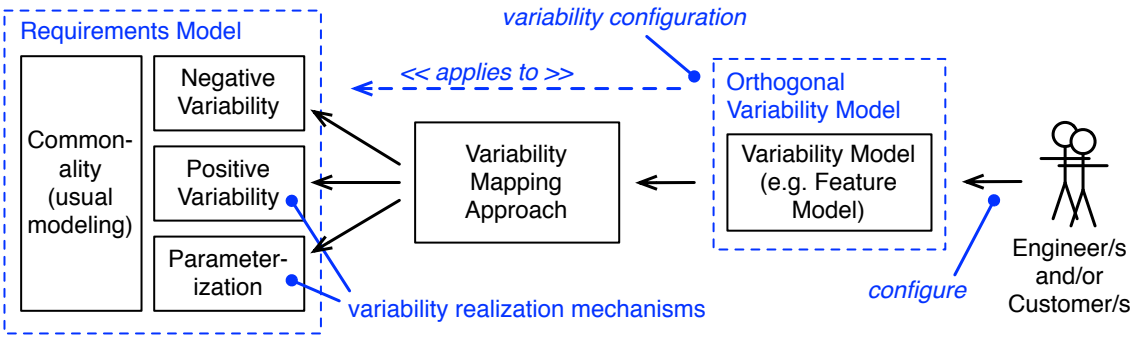


Figure 3.7: An overview of how variability configuration, variability model and the reference model connect via mappings and variability realization mechanisms.

relating variability to models

Heidenreich et al. provide a good general overview of how variability can be realized in models and introduced a classification of these variability mapping approaches [Heidenreich et al., 2010]. They consider only those approaches that use feature models as the (orthogonal) variability modeling notation, though. However, the presented mapping approaches could also be applied with any other variability modeling language. Figure 3.7 presents an overview of Heidenreich et al.'s summary on the state of the art of how variability models are related to other conceptual models of a software product line. As shown on the right hand side in Figure 3.7, engineers always configure only the variability model, in state-of-the-art approaches. All variable entities are then mapped to the other models of the software product line, where these variable model elements are realized as either negative variability, positive variability or via parameterization. With appropriate tool support such a constellation allows an automated derivation of product models that conform to the taken variability configuration (i.e., the selected and deselected variants or features, respectively).

variability realization mechanisms

Heidenreich et al. distinguished between three basic variability realization mechanisms [Heidenreich et al., 2010], as shown on the left-hand side in Figure 3.7. These work as follows:

- *Negative variability:* A comprehensive reference model is maintained, where all variable model elements are annotated with this respective variant (e.g., variable feature) of the variability model. When a product gets derived, all variants (e.g., variable features) that are deselected in the configuration get removed (i.e., pruned) from the model.
- *Positive variability:* A minimal reference model that contains only the elements common to all products is maintained and all variable model elements are defined as increments.

These increments are then mapped over to entities in the variability model. When a product gets derived, all increments that map to selected a variable entity are composed and all increments that map to a deselected entity are removed. Heidenreich et al. also mention the use of aspect models (i.e., aspect-oriented modeling) as an example and as a special case of positive variability [Heidenreich et al., 2010].

- *Parameterization of model elements:* A comprehensive reference model is maintained that already offers concrete model elements that can be modified to achieve variability. When a product gets derived, the values for the parameters are created by the mapping approach, which in turn takes the variability model's configuration as an input. An example for this approach would be component parameterization, where the behavior of the component varies depending on the parameter values.

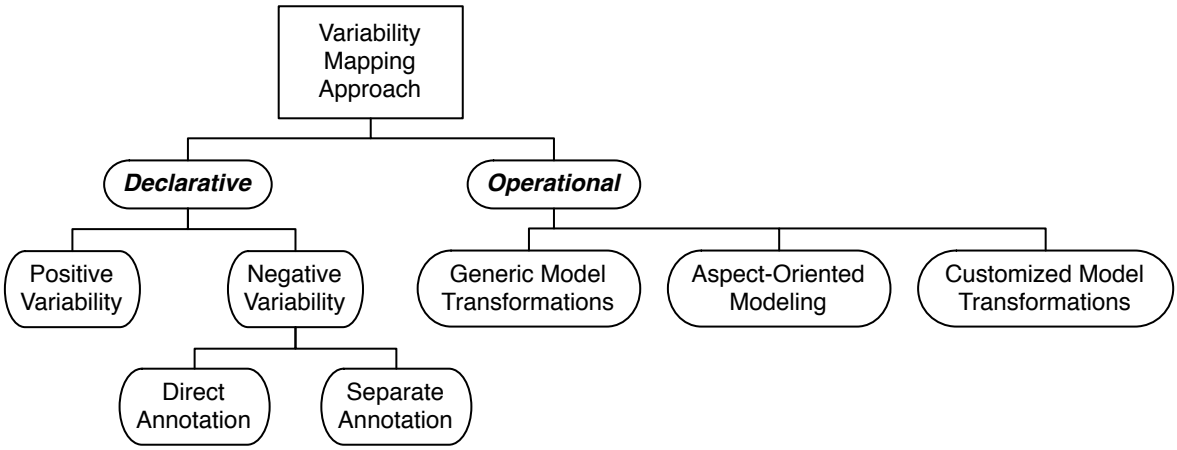


Figure 3.8: An overview of Heidenreich et al.'s classification of variability mapping approaches [Heidenreich et al., 2010].

Heidenreich et al.'s variability mapping approaches

Heidenreich et al. also introduced a more detailed classification of the actual mapping approaches used to link the orthogonal variability model and the variability realization mechanisms used in the requirements model [Heidenreich et al., 2010]. They generally distinguish between *declarative* and *operational* and between six concrete types of variability mapping approaches, see Figure 3.8. As 'declarative' they classify all approaches that only model *what* changes are needed but *not how* they need to be performed. The semantics of the actual variability realization mechanism is inherently encoded in the semantics of the model. These approaches include the AHEAD approach [Batory et al., 2004] and all annotative ones that use either direct annotation (e.g., model templates [Czarnecki and Antkiewicz, 2005] or the approach presented in [Morin et al., 2009]) or separate annotation (e.g., FeatureMapper [Heidenreich et al., 2008] or pure::variants [Beuche et al., 2004]). As 'operational' they classify

all variability mapping approaches that model *what* changes are needed and additionally also provide language constructs for specifying *how* they need to be performed. They distinguish three categories of approaches that use either generic model transformations (e.g., see [Ziadi and Jézéquel, 2006], [Botterweck et al., 2007]), aspect-oriented modeling (e.g., CVL [Haugen et al., 2008]) or customized model transformations (e.g., VML* [Zschaler et al., 2009], the approach presented in [Sánchez et al., 2009] or Gears [Krueger, 2012]).

Besides 'declarative' and 'operational' they also list other aspect-oriented approaches like MATA [Whittle et al., 2009], XWeave [Groher and Voelter, 2007] [Groher and Voelter, 2009], Reuseware [Heidenreich et al., 2009] and RAM (Reusable Aspect Models) [Kienzle et al., 2009] as possible variability realization mechanisms. They do not investigate these in more detail, but note that MATA, for example, does not provide support to automate the product derivation process, which means that the aspects corresponding to the selected features would need to be composed manually.

separate diagrams and mappings

Despite the short address of this strain of related work—a more detailed presentation would go beyond the scope of this thesis—the above remarks yet give a clear insight on how state-of-the-art approaches to variability modeling work. Namely, multiple separate diagrams are used to describe the requirements model and an additional orthogonal variability model together with a systematic mapping-based approach (i.e., as summarized above) are used to specify the commonality and variability. Furthermore, despite considerable work on the OVM and decision modeling, the majority of research in this area still tends to focus on *feature modeling*.

change sets and relationships

One other interesting paradigm that was not mentioned by Heidenreich et al. is Hendrickson and van der Hoek's approach to model product line architectures through change sets and relationships [Hendrickson and van der Hoek, 2007]. Their core idea is to specify 'features' as change sets, which include a detailed description of additions and removals of required architectural components (i.e., related architectural differences) in the model. Further, they use relationships to define which change set combinations are valid. This way Hendrickson and van der Hoek's approach is novel as it does not separate the variability model and the software architecture model as other existing approaches do (i.e., the variable entities and their actual realization). This concept allows a very dynamic application of any change sets at any time and still yields a valid architectural model for any such change. Compared to their traditional architectural product line modeling solution, which builds on xADL 2.0 (see [Garg et al., 2003]), change sets and relationships have turned out to be a much more intuitive, efficient and compact notation. This concept could, thus, also be promising when realized with requirements modeling in a similar manner. Their approach has also inspired some of the ideas presented in this thesis, as introduced in Part II. Currently, Hendrickson and van der Hoek's approach is limited to software architecture modeling, though.

3.3 Automated Variability Analysis

why? Manually checking whether a full product configuration satisfies all constraints can be very laborious and non-trivial, once variability models grow large and complex. Batory et al. argue that automatically validating and analyzing product specifications will have significant practical payoffs, which motivates the research in this area [Batory et al., 2006]. They envision tools that propagate constraints (so that incorrect specifications can automatically be detected), that provide explanations when dead ends in the design are reached (and solutions to fix such designs as well), and that automatically optimize configurations for specific needs (to simplify program designs). Although Batory et al. focus on feature models only, their mentioned challenges can be regarded as valid for variability modeling languages in general.

what gets analyzed? Batory et al.'s following list of challenges for the automated analysis of feature models provides a refined list of the abstract goals that automated analysis of variability models aims to achieve [Batory et al., 2006]:

- *Model Consistency:* Some industries use feature models with thousands of features (e.g., see [Reiser and Weber, 2006]) and features often need to satisfy Boolean or numerical constraints. Batory et al. argue that it is well known that these models are riddled with inconsistencies that are difficult to detect. Since the way these inconsistencies are discovered is mostly accidental, today, the goal is to develop automated ways and tool support to find such inconsistencies.

- *Explanations:* When a model has an inconsistency it is often not trivial to find out why this model is inconsistent and how the inconsistency can be rectified. Thus, another general goal is to have tools that, for example, find the minimal number of violated constraints for an inconsistency, or that suggest minimal sets of changes of the existing constraints to fix the inconsistency.

- *Variability in Model-Driven Development (MDD):* When dealing with variability in MDD it is fundamental to map the variability model to the development artifacts (e.g., requirements, architecture, code modules, test cases, documentation), recall Section 3.2.2. Batory et al. noted that verifying whether other program representations are consistent with their feature model is a significant research challenge. Thus, the general goal with regards to model-driven development is to develop tool support that can automatically verify whether other representations of a product are consistent with their fully configured product variability model.

- *Multi Product Lines:* Batory et al. highlight that there are real mega-projects like product lines of aircraft carriers that, for example, carry different types of airplanes which are themselves developed as product lines, too. This leads to more complex constellations which others also called nested product lines [Krueger, 2006], dependent product lines

> [Rosenmüller et al., 2008] or multi product lines [Rosenmüller and Siegmund, 2010] [Dhungana et al., 2011]. Also, reuse across several software product lines should be considered in this context [van Ommering, 2002]. The general goal in this area is, as stated in [Batory et al., 2006], to generalize variability models, to describe such "mega" products, and to develop tool support that allows analyzing and visualizing these models.

and how?

The above gives a good overview on the general challenges and goals of automated variability analysis. Batory et al., however, already listed one more challenge that was more about *how* to achieve the above, namely *performance scalability* [Batory et al., 2006]. There are various automated reasoning concepts today, like Boolean satisfiability (SAT) solvers, binary decision diagram (BDD) solvers, constraint satisfaction problem (CSP) solvers, artificial intelligence (AI) configurators, symbolic model verifiers (SMV) or description logic-based reasoners. Benavides et al. provide a classification of these, as follows in Section 3.3.1 [Benavides et al., 2010]. Batory et al. highlighted that despite the enormous increase in computing power the problems of feature combinatorics still remain NP-hard. Since not all tools and approaches will perform equally well, the identification of which tools to use when and whether a combination or integration of several solvers makes sense is also considered a research challenge [Benavides et al., 2010].

the remainder of Section 3.3

There is a plethora of sometimes quite diverse approaches for the automated analysis of various variability modeling languages. Most of these automated analysis approaches build on feature modeling, or on a specific dialect thereof. Fortunately, Benavides et al. have already presented a quite comprehensive state-of-the-art survey on automated analysis of feature models quite recently [Benavides et al., 2010]. In the following, we will therefore first briefly report on the key insights from [Benavides et al., 2010]'s survey in Section 3.3.1, which we supplemented with an additional illustration of key activities. For AND/OR tables there is no similar automated variability analysis approach, since they do not support any systematic specification of variability constraints. However, an automated analysis to discover hierarchies, feature groups, and default settings could also be done with AND/OR tables [Czarnecki et al., 2008]. Further, for decision modeling and OVM we also briefly discuss the most important automated analysis approaches in Section 3.3.2. The latter two do not only analyze the variability model, but also include the requirements model in the automated analysis. The lion's share of existing work on automated analysis of variability models is based on feature models, though, as follows.

3.3.1 Automated Analysis of Feature Models

types of feature models

Benavides et al. have recently presented a comprehensive survey on the state of the art of the automated analysis of feature models [Benavides et al., 2010]—twenty years after the original introduction in [Kang et al., 1990]. Various variability modeling dialects have been developed over the past twenty years and there exists no standardized notation today. Benavides

et al. distinguish between three major types of feature models: (i) basic feature models, (ii) cardinality-based feature models, and (iii) extended feature models. Basic feature models are defined as introduced in Section 3.2.1 and allow only simple *requires* and *excludes* constraints. Cardinality-based feature models additionally introduce minimum and maximum cardinalities for both features (i.e., where features can also be instantiated multiple times) and group features (i.e., not only or [1..n] and xor [1..1], but any arbitrary cardinalities). Extended feature models allow the definition of attributed features (i.e., features that also have at least a data type and a value in addition to their name) on which so-called extra-functional features can be defined. Benavides et al. define extra-functional features as relations between one or more attributes of a feature, which can be written in arbitrary forms and can also include classic mathematical operators like $+$, $-$, $*$, $/$, $<$ or $>$ [Benavides et al., 2005].

General Process for the Automated Analysis

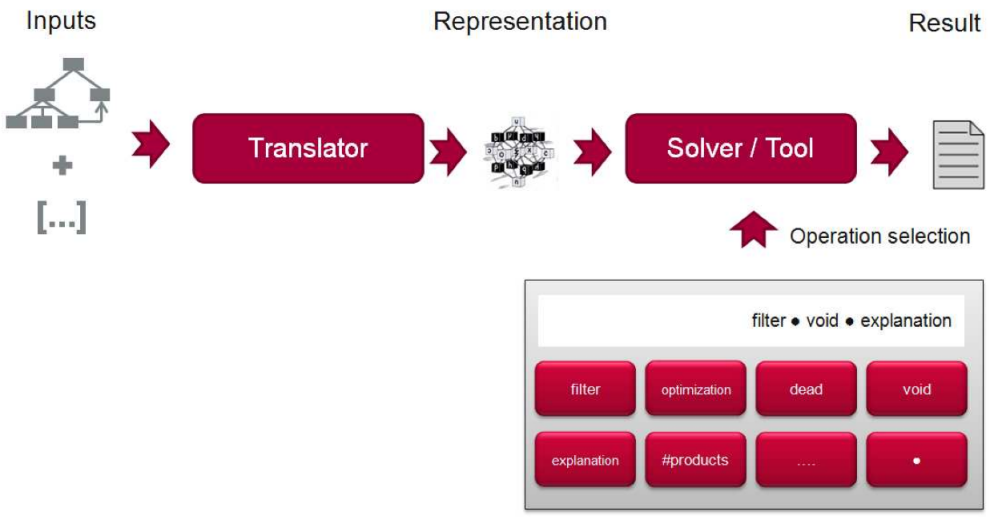


Figure 3.9: Benavides et al.'s general process for the automated analysis of feature models, taken from [Benavides et al., 2010].

In their survey Benavides et al. investigated 53 papers as primary studies and found that the terms used were usually ambiguous [Benavides et al., 2010]. Therefore, they proposed a conceptual framework to provide a high-level vision of the analysis process that subsumes all the common concepts and practices found in their primary studies. This abstract conceptual framework clarifies the general, abstract process, see Figure 3.9. A very similar process was also shown earlier in a survey on formal methods in software product lines, however, see [Janota et al., 2008, Fig. 5].

While this process is very abstract, it shows the general idea. The variability model is the input and gets parsed into an equivalent definition in a formal notation (called *Representation*) and this formal definition gets analyzed automatically by a *Solver / Tool*, which then generates the analysis *Result*. Not shown in this general process is that the *Tool* used typically uses an existing off-the-shelf solver or reasoning tool only as a bottom-end, but often runs dedicated higher-level algorithms itself. Also, the results of the automated analysis are typically presented in the context of the variability model, too. This would result in a feedback loop of the *Results* back to the *Inputs*, which is not visualized in Figure 3.9. The definition of the automated analysis operation (i.e., the *Operation selection*) typically also requires the *Inputs* and/or previous results for many particular operations, as follows.

Translation into a Formal Representation

Because it is crucial for understanding Part II of this thesis, a short example of how the *Input* can be *translated* into a formal *Representation* is provided here (Figure 3.9). [Mannion, 2002], [Zhang et al., 2004], [Batory, 2005] and [Bontemps et al., 2005] were among the first to show how a feature model can be translated into a propositional logic form, which allows the use of, e.g., SAT, BDD, SMV or CSP solvers. Asikainen et al. also presented a solution to translate feature models into another machine-readable format and to automatically analyze them with an AI configurator that was previously developed for use in a non-software context [Asikainen et al., 2004a]. To demonstrate how this translation works, however, we chose a simple but yet concise example from [Czarnecki and Wasowski, 2007] for an illustration.

from feature diagrams to logic – an example

Figure 3.10 illustrates how an example feature model can be translated into an equivalent Boolean formula in propositional logic—this example is borrowed from [Czarnecki and Wasowski, 2007]. The upper part of Figure 3.10 shows the feature diagram already used in Figure 3.3 and an explanation of the syntax in blue color. The lower part shows an equivalent and straight-forward representation of this feature model as a Boolean formula. Features are abbreviated with single letters, as underlined in the diagram above. The formula consists of five basic relationships between features with specific formal semantics: A *child-parent* relationship specifies that whenever a child feature gets selected also its parent feature needs to be selected. A *mandatory* feature requires that whenever a parent feature is selected and has a mandatory child feature, this child feature needs to be selected as well. An *or-group* requires that whenever the parent feature of the group constraint is selected, at least one of the features, that are part of the or-group, also needs to be selected. A *xor-group* requires that whenever the parent feature of the group constraint is selected, then exactly one of the child features needs to be selected. Finally, the *additional* constraints are already defined in a propositional form and merely need to be added to this set of constraints as well. The logical formula at the bottom of Figure 3.10, hence, is the semantic equivalent to the actual feature diagram above. Importantly, this translation of any feature model into such an equivalent formula in Boolean logic can quite easily be

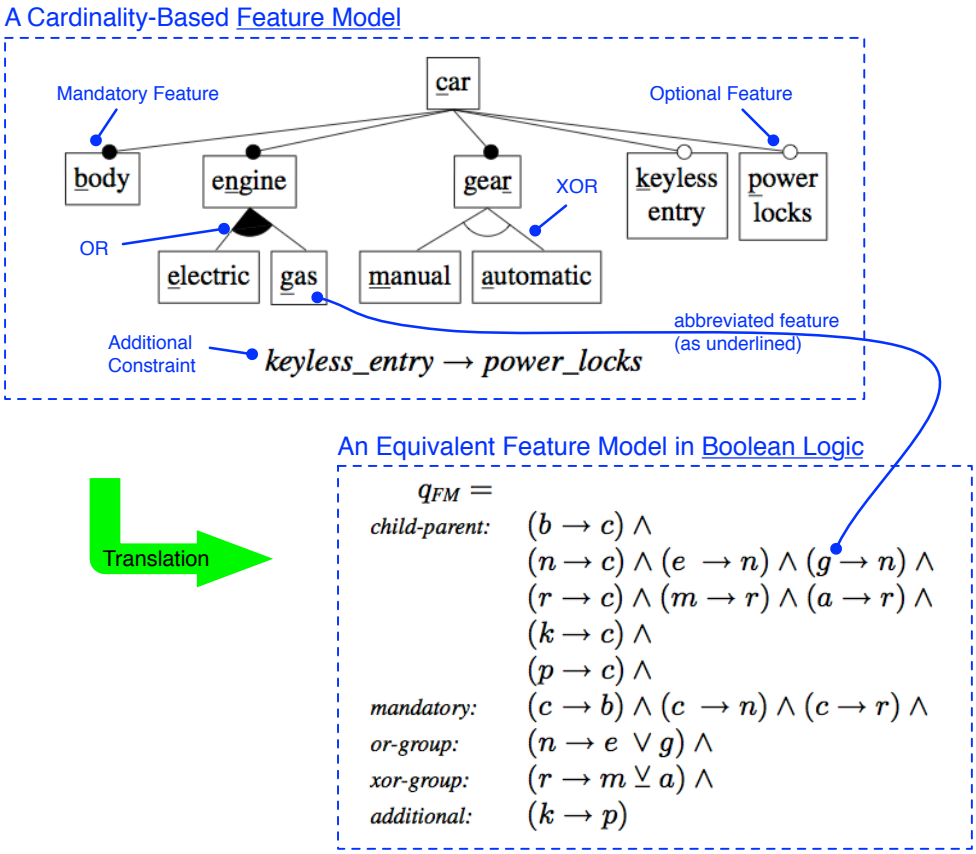


Figure 3.10: A cardinality-based feature model and its representation in Boolean logic, taken from [Czarnecki and Wasowski, 2007].

automated. Based on such a translation Czarnecki and Wasowski applied a BDD solver for automated variability analysis [Czarnecki and Wasowski, 2007]. However, this formula can also straightforwardly be translated into a conjunctive normal form (CNF) and can also be analyzed with a SAT solver—see [Batory, 2005], for example.

Automated Analysis Operations

Numerous operations for automated analysis have been proposed. In their survey Benavides et al. identified 30 different automated analysis operations in the current literature [Benavides et al., 2010]. Table 3.1 gives an overview of these. In the following we briefly introduce the three most important ones in the context of this thesis, with a focus on constraint propagation. For an explanation of the others please refer to [Benavides et al., 2010, Chapter 5].

Table 3.1: Benavides et al.'s thirty automated variability analysis operations, as listed in [Benavides et al., 2010].

1	Void feature model	*11*	Valid partial configuration	*21*	Conditional dead features
2	# Products	*12*	Atomic sets	*22*	Homogeneity
3	Dead features	*13*	False optional features	*23*	LCA
4	Valid product	*14*	Corrective explanations	*24*	Muti–step configuration
5	All products	*15*	Dependency analysis	*25*	Roots features
6	Explanations	*16*	ECR	*26*	Specialization
7	Refactoring	*17*	Generalization	*27*	Degree of orthogonality
8	Optimization	*18*	Core features	*28*	Redundancies
9	Commonality	*19*	Variability factor	*29*	Variant features
10	Filter	*20*	Arbitrary edit	*30*	Wrong cardinalities

automated analysis operations

Most of the operations in Table 3.1 are quite simple to understand, once the general concept of automated analysis of feature models has been understood. For example, *void feature model* checks whether there is a contradiction among the constraints in the feature model—this would lead to unsatisfiability, respectively zero possible valid products, and, thus, a void feature model. Or *# products* iterates through all possible product configurations, checks the satisfiability of every of them and counts the number of all valid products, hence. Von der Maßen, for example, has called this operation the *variation degree* and presented it in more detail [von der Maßen, 2007]. Some other operations are slightly more complex, like *multi-step configuration*, for example. Benavides et al. derive this operation from [White et al., 2009]'s work, where multi-step configuration targets at situations where single products are iteratively developed over large time periods (i.e., several years). An initial version, thus, implements only few features and later versions add additional features until the final configuration is reached. The analysis operation calculates a set of consistent, intermediate product configurations based on the feature model, the initial configuration, a desired final configuration, a number of steps in the configuration path, a global constraint that must not be violated (i.e., which usually refers to feature attributes), and a function determining the cost to transition from one configuration step to another one.

product derivation

While Benavides et al.'s paper gives a good general overview [Benavides et al., 2010], it does not provide a good insight into today's state of the art in product derivation—i.e., the process of incrementally configuring a domain feature model into a full and valid configuration of a concrete product. Benavides et al. distinguished between full configurations (where all features are bound to either selected or deselected) and partial configurations (where only some features are bound and some yet explicitly left undecided). Nevertheless, they do not present important operations like Czarnecki et al.'s staged configuration [Czarnecki et al., 2005a] or Batory's Boolean constraint propagations [Batory, 2005]. These operations are yet more complex, but crucial for providing sophisticated automated analysis support during a product derivation. Since the concept of constraint propagation is crucial for understanding Part II of this thesis, we briefly illustrate the state of the art in this area in the following.

staged configuration

Czarnecki et al. motivate that staged configuration is important in a realistic development process where different groups and different people make product configuration choices in different stages [Czarnecki et al., 2005a]. Czarnecki et al.'s staged configuration can be achieved either by stepwise specialization of a feature model or by multi-level configuration, as follows.

As *stepwise specialization* Czarnecki et al. defined *"the transformation process that takes a feature diagram and yields another feature diagram, such that the set of the configurations denoted by the latter diagram is a subset of the configurations denoted by the former diagram"* [Czarnecki et al., 2005a]. In other words, a specialization of a feature diagram is the binding of variability (i.e., the selection, deselection or narrowing of cardinalities in a cardinality-based feature model), which leads to the generation of a refined feature model with less variability remaining. As presented in [Czarnecki et al., 2005b] such a specialization step also refines feature cardinalities and group cardinalities, removes sub-features from a group, and so forth, in such a way that the resulting feature model of a specialization step has no inconsistencies with respect to its constraints. This automatic processing of a specialization step is similar to Batory's *Boolean constraint propagation* [Batory, 2005]. A fully specialized feature diagram exhibits a specific (full) *configuration*, where all variability constraints are satisfied and no variability is left undecided (a configuration in [Czarnecki et al., 2005a], thus, is always a full configuration as in [Benavides et al., 2010]).

As *multi-level configuration* Czarnecki et al. further define the process of taking all feature decisions on different hierarchy levels (e.g., on product-line level or system level) of the feature diagram subsequently, starting from the highest hierarchy level and descending [Czarnecki et al., 2005a]. This way the configurations previously taken in more abstract levels can be used to automatically specialize the more fine-grained feature models in the later levels. Such automatic specialization also ensures consistency between configurations in different levels. This process continues until a consistent full configuration has been reached. Multi-level configuration, hence, avoids conflicts that emerge when different stakeholders work on different abstraction levels and their configurations become inconsistent with each other.

Generally, Czarnecki et al. argue that breaking up the modeling task into such separate levels helps to define a more clear focus on what type of features a level is allowed to have [Czarnecki et al., 2005a]. Such focus helps to avoid the so-called "analysis paralysis", which is the problem of not knowing when to stop adding more detailed features.

constraint propagation

To ensure that the selection of desired features from the feature model is *"within the variability constraints defined by the model"* (see [Czarnecki et al., 2005b]) a realization of what [Batory, 2005] and we call *constraint propagation* is necessary every time a stakeholder takes a variability binding decision (i.e., performs a specialization operation). Batory has described the idea of considering a feature model as a logic-truth maintenance system (LTMS) and of realizing Boolean constraint propagation on that basis [Batory, 2005]. He motivates that *"as users select features for a desired application, we want the implications of these selections to be propagated, so users cannot write incorrect specifications."* Batory introduced a three-value logic (i.e., using

the values $true$, $false$, and $unknown$) for variable features, where a feature can also explicitly be *undecided*, or respectively $unknown$—these two terms have the same meaning. Therefore, any feature that is neither selected nor deselected is set as $unknown$. The concept of constraint propagation is crucial for systematically supporting product derivation with automated variability analysis. Batory suggested the use of off-the-shelf SAT solvers to perform this calculation [Batory, 2005]. In order to illustrate the basic idea of *constraint propagation* we use an example from [Czarnecki et al., 2005b]. Note that stepwise specialization as in [Czarnecki et al., 2005b] and Boolean constraint propagation as in [Batory, 2005] are quite similar concepts, though.

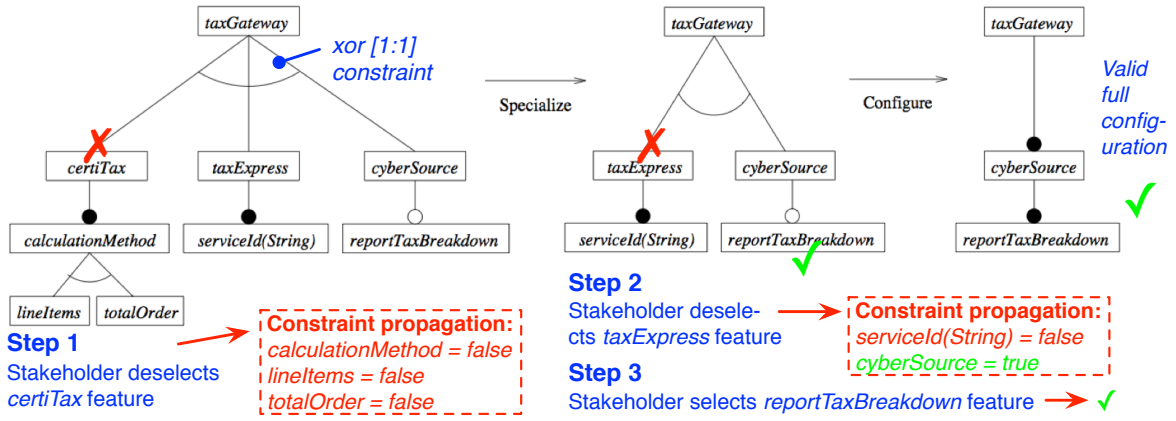


Figure 3.11: Constraint propagation highlighted and illustrated in a feature model example from [Czarnecki et al., 2005a], which illustrated specialization and configuration.

example and illustration

Figure 3.11 presents an example feature model of tax gateways, which is part of a product line of electronic shops [Czarnecki et al., 2005a]. The annotations added in Figure 3.11 highlight where a constraint propagation happens in this example configuration, taken from [Czarnecki et al., 2005a], which guarantees that every of the shown models is consistent. In *Step 1*—see Figure 3.11—a stakeholder has decided to deselect the feature *certiTax*. This requires that all sub-features of this feature are also deselected, which is the constraint propagation of this deselection (see the red text framed with a dashed red line in Figure 3.11). The result is a specialized feature model where *certiTax* and its sub-features are no options anymore. The specialized feature model is again a fully valid model, though—see the diagram in the middle of Figure 3.11. Now a stakeholder may decide to deselect the feature *taxExpress* in *Step 2*, which also requires a deselection of its sub-feature and a selection of the *cyberSource* feature, which is the third feature that is part of the xor constraint. Thus, deselecting *taxExpress* needs two propagations. The resulting feature model of this specialization is not shown in Figure 3.11, though—it would be the same model as on the right-hand side, but with feature *reportTaxBreakdown* still optional. Finally, in *Step 3*, this feature *reportTaxBreakdown* may become selected, which requires no

constraint propagations anymore and leads to a full configuration, as shown on the right-hand side of Figure 3.11.

In summary, this example highlights Batory's idea of Boolean constraint propagation [Batory, 2005] in the context of Czarnecki et al.'s stepwise specialization [Czarnecki et al., 2005b]. It shows the similarity of the two and also introduces a crucial automated analysis operation for product derivation, which we call *constraint propagation* (i.e., as follows later in Chapter 8). The example further shows how an equivalent feature model gets generated for a partial configuration—this capability is already supported in some commercial tools like, e.g., pure::variants from pure-systems GmbH. Benavides et al.'s survey only considered specialization (a concept similar to constraint propagation) as a special case of an edit operation in a domain feature model (i.e., as introduced by [Thüm et al., 2009]) and mentioned the concept of constraint propagation only briefly as *dependency analysis* [Benavides et al., 2010].

Groups of Approaches for Automated Support

various automation approaches

Benavides et al. classify the underlying automation support into three dedicated groups, which use the following representations and types of solvers [Benavides et al., 2010]:

- *Propositional logic based analyses:* Benavides et al. define a propositional formula to consist of a set of primitive symbols or variables and a set of logical connectives constraining the values of the variables, e.g., $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$. Benavides et al.'s survey reveals that the most used tools in this category are SAT solvers and BDD solvers, but also others like Alloy [Jackson, 2006] or SMV have been proposed. Hitherto results have shown that SAT bears the best average performance for large models, but that BDD is yet faster for some specific operations like counting the number of all products, for example.

- *Constraint programming based analyses:* Benavides et al. define that in contrast to propositional formulas, constraint satisfaction problem (CSP) solvers can also deal with numerical values such as integers or intervals. This allows including concrete values of feature attributes of extended feature models into the automated analysis. Xiong et al. show how such non-Boolean models can also be handled by SMT solvers [Xiong et al., 2011]. Benavides et al.'s survey reveals that the most used tools in this category are JaCoP, Choco, OPL studio, and GNU Prolog (see [Benavides et al., 2010] for references)—in this order.

- *Description logic based analyses:* Benavides et al. define description logics as a family of knowledge representation languages which use a set of concepts (a.k.a., classes), a set of roles (e.g., properties or relationships), and a set of individuals (a.k.a., instances) to describe a problem. For example, OWL-DL (the web ontology language-description logic, standardized by the World Wide Web Consortium) has been used to express feature

models and the tools used in this category are, for example, RACER, the OWL debugging tool or Pellet (see [Benavides et al., 2010] for references).

Benavides et al. also identified numerous studies that rely on the use of ad-hoc algorithms and representations for the automated analysis of variability models—other than the above mentioned [Benavides et al., 2010]. They presented an overview of the 53 primary studies they surveyed and classified every of them into the three categories above. Some of these studies also use multiple solvers. Some of them use none of these solvers, but propose their own tools. Some do not even provide any automated support. The lion's share, however, are those approaches that use an underlying automated analysis tool that falls into one of the above three classes.

Other Observations

interactive configuration

Janota has presented a comprehensive realization of Batory's logic-truth maintenance system (LTMS) [Batory, 2005] to realize interactive configuration for feature models, based on SAT solving [Janota, 2008] [Janota, 2010]. In terms of automated support for product derivation (i.e., considering only the configuration of the feature model) this reference may be the most advanced and sophisticated one. Interactive configuration in [Janota, 2010] calculates the propagation of implications of user decisions by algorithmic exploration for conflicts and backtracking. Janota additionally shows how *explanations* for automatically selected decisions can be visualized in the feature model, which help a user to understand which of his or her previously taken feature selections or deselections caused another decision to be set automatically because of the feature model's constraints [Janota, 2010, pp. 135].

constraint relaxation vs. corrective explanations

Janota also discussed the concepts of *constraint relaxation* and *corrective explanations*, which are possible solutions for situations where a user decision violates the configured problem [Janota, 2010]. When a configuration does not satisfy all constraints, one can either decide (i) to relax the constraints that are involved in the conflict or (ii) to change the actual configuration as slightly as possible such that all constraints are again satisfied. The first strategy goes slightly into the direction of living with inconsistencies in configurations [Wang et al., 2010]. The second strategy is a more sophisticated solution for interactive configuration, which automates backtracking and an exploration of similar but consistent configurations. An actual solution for automatically calculating such corrective explanations has not been provided in [Janota, 2010], though.

Papadopoulos and O'Sullivan presented an algorithmic solution that automatically calculates and suggests such constraint relaxations for constraint satisfaction problems in general [Papadopoulos and O'Sullivan, 2008]. This could be used to realize the first strategy in cases where the engineers really desire a specific but unsatisfiable configuration. O'Callaghan et al. showed how corrective explanations can be realized with constraint satisfaction problems in general

[O'Callaghan et al., 2005]. In earlier work on interactive configuration in classic engineering Felfernig et al. also proposed a similar solution based on CSP solving [Felfernig et al., 2001]. While their goal was to reach constraint satisfaction, they did so by relaxing the unary user constraints (i.e., the hitherto taken configuration decisions). Because these only reflect the hitherto taken configuration decisions and *not* the actual configuration problem, their approach also falls into the category of corrective explanations. Later-on White et al. presented a very similar solution and showed how such corrective explanations can be performed with specific flawed feature model configurations [White et al., 2008]. Nöhrer and Egyed further explored strategies that temporarily allow inconsistency and aim at resolving these conflicts only late in the configuration process [Nöhrer and Egyed, 2010]. Similarly, Wang et al. argued that it is difficult, in general, to always ensure consistency when constructing feature models [Wang et al., 2010]. Thus, they argued for temporarily tolerating inconsistencies (i.e., contradictory constraints) and detecting and fixing these only later (i.e., inspired by Balzer's work on tolerating inconsistency [Balzer, 1991]). Wang et al. developed their own constraint solver for this purpose, which distinguishes between consistent and inconsistent parts of a feature model and, thus, still allows an automated analysis of the consistent parts, while there is inconsistency [Wang et al., 2010].

diagnosing feature model errors

Trinidad et al. further argue that when using agile methods and software product line engineering together, feature models are crucial and strongly affected by changes on requirements (which are introduced very frequently in agile approaches) [Trinidad et al., 2008]. Therefore, they argue that changing large-scale feature models as a consequence of changes on requirements is a well-known and error-prone activity. Hence, they aim at fixing errors that already exist, rather than avoiding such errors already in the first place. Their approach builds on Reiter's theory of diagnosis [Reiter, 1987] and CSP solving and provides the engineers with a list of errors and explanations in the feature model. In contrast to [Trinidad et al., 2008]'s work, which deals with domain feature models (i.e., where no selection or deselection was done yet), White et al. continued this strain of work and proposed approaches to fixing erroneous feature configurations (i.e., where given selections and deselections do not satisfy the constraints) with corrective explanations [White et al., 2008] [White et al., 2009]. These approaches do not realize the variability modeling tool as a logic truth maintenance system (LTMS), though, as suggested by Batory [Batory, 2005], but rather allow errors to occur in feature models and provide tool support for diagnosing and correcting them later-on.

3.3.2 Automated Analysis of Requirements and Variability

The previous Section 3.3.1 has presented the landscape of existing automated analysis approaches for feature models. Most of these approaches actually focus on feature models only, which implies that the involved requirements specifications can still be erroneous because they are not included in the verification. But there are extensions that also cover requirements specifications.

The subsequent sub-sections give a brief overview on approaches that automatically analyze both the variability model and the requirements specification. These include approaches based on decision modeling, feature modeling with OCL, OVM, and feature modeling with behavioral models, chronologically ordered with the earliest work first. Interestingly, the body of knowledge in automated analysis on decision models and OVM is rather limited and these approaches tend to combine the verification of variability *and* requirements much more than feature modeling-based work. The typical automated analysis operation we found here is consistency checking.

Decision Model and Textual Requirements

verifying textual product line requirements

DECIMAL (DECIsion Modeling AppLication) [Padmanabhan, 2002] [Padmanabhan and Lutz, 2005] was one of the first approaches that explicitly proposed an automated analysis of variability modeling. Their novel automated analysis solution was motivated by the fact that *"there is currently no automated way to verify the completeness and consistency of the new product's requirements in terms of the product line"* [Padmanabhan and Lutz, 2005]. Padmanabhan and Lutz's solution does not provide a full consistency checking of the product line model itself, but rather verifies whether a derived product requirements specification is consistent with all constraints defined in the product line specification [Padmanabhan and Lutz, 2005]. Padmanabhan's decision modeling notation is similar to the one in [Schmid and John, 2004], but DECIMAL does not describe a mapping to various other software artifacts (e.g., that are extended with specific variability mechanisms as in [Schmid and John, 2004]), but rather provides an integrated solution for textual requirements specification and decision modeling [Padmanabhan, 2002].[2] Constraints in DECIMAL (e.g., dependency relationships among variabilities) are directly specified in predicate logic (i.e., first-order logic). Padmanabhan's automated analysis solution, however, was yet quite simple: they store the whole decision model in a database system and use SQL to perform queries, where they parse relevant constraints into *condition* statements such that all returned results are assured to satisfy these constraints. Their tool automatically checks (i) the completeness and consistency between a new product and the product line and (ii) that dependency constraints (e.g., choices of features) are satisfied. They also describe how to construct feature arbitration policies (i.e., additional constraints) to avoid unwanted feature interactions, which are then checked automatically in the same way.

While Padmanabhan's work was one of the first (or possibly the first) to automatically analyze variability constraints in a software product line, the solution yet had many problems. For example, when constraints transitively impact each other, this will not be considered in the results. Also, the scalability of their SQL-based solution (they did not present a performance evaluation) is probably much weaker than a solution that directly builds on modern SAT or CSP

[2] This integration is similar to how feature modeling and variability modeling are integrated in a today's *feature modeling* notations, since [Kang et al., 1990]—see Section 3.2.1.

solvers. Such a solution has more recently been presented by Mazo et al. for the automated analysis of decision models in the DOPLER notation (cf. [Dhungana et al., 2010]), see [Mazo et al., 2011]. This solution uses the GNU Prolog constraint solver [Diaz and Codognet, 2001]. Mazo et al.'s work considers the decision model only, however, and does not include any other conceptual models in the automated analysis [Mazo et al., 2011] (decisions in DOPLER can also be regarded as textually specified requirements).

Feature Models and UML with OCL

Czarnecki and Pietroszek were possibly the first to introduce an approach that checked the consistency of requirements of an entire product line against the variability model (i.e., for all products) [Czarnecki and Pietroszek, 2006]. The basic idea was to verify that the model of each product, which corresponds to a correct configuration, is well-formed. Well-formedness in this context refers to the satisfaction of any types of static constraints that can be defined over the product models. The approach can be used for requirements models, design models and implementations. The paper shows examples that use feature modeling and UML. The defined approach works at the level of the meta model of the target language, however. Thus, any other modeling language that has a meta model and the desired constraints written in the Object Constraint Language (OCL) can also be used. These desired constraints include the constraints that are part of the meta model, typing rules, and architecture-specific rules. An example of the latter would be that there should be no more than one component of some type. These constraints are formulated in OCL, for the concrete modeling language used, to express the product models. While the approach was presented with feature models it could easily be adapted to other variability modeling languages as well.

OVM and Behavioral Models

model checking entire product lines

In the context of OVM [Pohl et al., 2005], Lauenroth and Pohl introduced an approach that focuses on the automatic consistency checking of product line requirements specifications as a whole, which verifies that all derivable products are consistent [Lauenroth and Pohl, 2007]. Similarly, also Fantechi and Gnesi present an approach that uses labels to annotate variable elements in labeled transition systems (LTS) [Fantechi and Gnesi, 2007]. However, they did not introduce any additional, orthogonal variability modeling language and also no tool support for automated analysis or verification, yet. In further work, Fantechi and Gnesi sketch the possibility that properties verified in the family are preserved by the derivation of products [Fantechi and Gnesi, 2008].

OVM, on the other hand, relies on a variability mapping approach with *negative variability*, recall Section 3.2.2, which implies that the product line requirements specification (PLRS) is a

comprehensive reference specification that includes all requirements—even those contradictive with each other. Therefore, the product line requirements specification may be inconsistent, when building on comprehensive reference models with separate annotation of variability (Section 3.2.2) and when variability constraints are not explicitly taken into account.

Lauenroth and Pohl presented a general framework for automated consistency checks for static properties of product line requirements specifications, which allows verifying that no contradicting requirements are defined in a product line [Lauenroth and Pohl, 2007]. Later-on they further presented an approach based on this framework for checking the consistency of behavioral invariants in product line specifications [Lauenroth and Pohl, 2008]. Furthermore, they presented a solution that is built upon computation tree logic (CTL), which supports the model-checking of even richer property specifications [Lauenroth et al., 2009]. In summary, Lauenroth et al.'s approach allows verifying the fulfillment of CTL properties for any variable I/O-automaton as a whole. Therefore, in general, their model checking solution allows verifying that every possible product that can be derived from a domain artifact fulfills the specified properties.

Feature Model and Behavioral Models

model checking lots of systems

While all solutions presented in Section 3.3.1 have focused on feature models *only* (Section 3.3.1), Classen et al. combined the automated analysis of behavioral models with the variability specified in feature models [Classen et al., 2010]. Not combining the analysis of the two would leave the possibility that the behavioral requirements specification is erroneous, since the mappings (recall Section 3.2.2) and the consistency of all the other software artifacts were not included in the automated analysis. Classen et al. defined and implemented a model checking technique that allows verifying such transition systems (i.e., the behavioral specification of an entire system family) against temporal properties [Classen et al., 2010]. They provide a model checking approach that is capable of verifying a product line specification as a whole. The approach allows verifying all the products of a product line at once and to pinpoint the products that violate particular properties. In [Classen et al., 2011] the authors continue to study symbolic algorithms, rather than explicit ones, which allows the further handling of much larger state spaces. Their approach for symbolic model checking builds on an industry-strength symbolic model checker called NuSMV [Classen et al., 2011].

CHAPTER 4

Open Problems

UML and orthogonal variability

Today's state-of-the-art languages in requirements modeling (recall Chapter 2) were designed for modeling classic, monolithic software systems. The UML, for example, has never explicitly introduced the modeling of commonality or variability as a fundamental concept of the language. Describing the model with many separate diagrams, as the UML does, has often been perceived to ease early-phase requirements modeling activities. However, this comes at the cost of consistency and comprehensibility of the model as a whole—recall Section 2.2. While the UML's fundamental design has become established and is well suited for single system modeling, we argue that it becomes disadvantageous when modeling software product lines. Creating and maintaining several diagrams to represent the requirements model leaves no other choice than adding an additional, orthogonal variability modeling notation and a sophisticated variability mapping approach to specify the variability—recall Section 3.2. While there are numerous such mapping-based approaches, they inevitably lead to inefficiencies in comprehensibility of the product line requirements model. Variable features are not treated as first-class entities in such requirements models, which rely on orthogonal variability modeling.

open problems

We argue that orthogonal variability modeling is disadvantageous because variable features should be first-class entities in the conceptual requirements modeling language and notation (recall Section 3.1). The split of the product line requirements model into multiple separate diagrams (i.e., various UML diagrams and the variability model) leads to the following three major problems.

4.1 Information Scattering

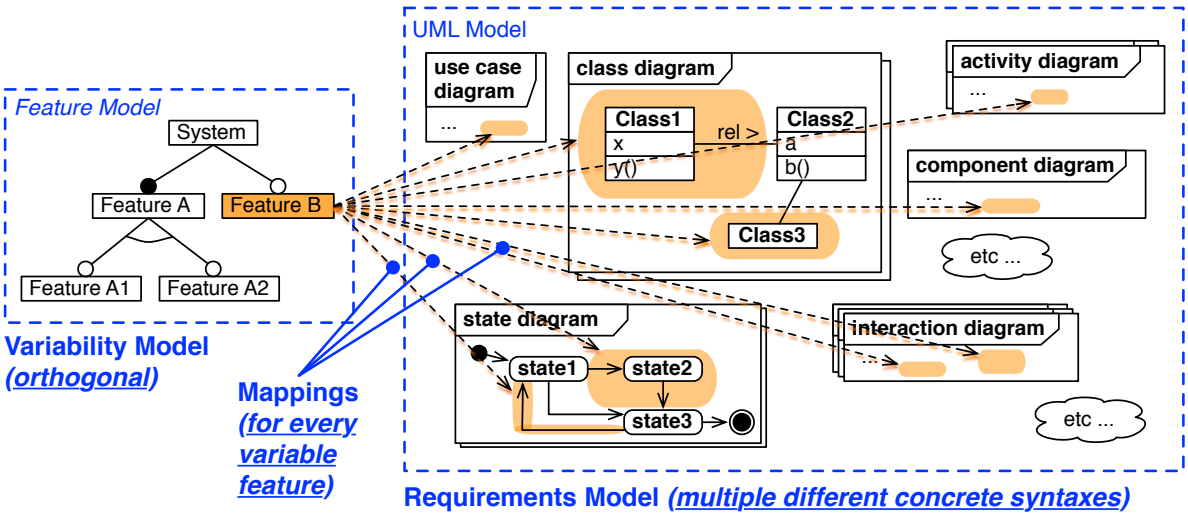


Figure 4.1: A conceptual overview of state-of-the-art requirements and variability modeling (UML and feature modeling) with the scattered specification of *Feature B* highlighted in orange color.

information scattering by example

Existing variability mapping approaches (recall Section 3.2.2) lead to an information scattering of the specification of a variable feature over multiple different diagrams. Figure 4.1 visualizes this information scattering of a variable feature's specification in a typical state-of-the-art modeling scenario (e.g., that relies on the UML and feature modeling). Figure 4.1 shows a generic variability model in the form of a feature model on the left-hand side and a UML model as a collection of many separate UML diagrams on the right hand side. A typical variable feature in a software product line impacts multiple facets of the software product (e.g., structure, data, behavior, user interaction, etc.) and, thus, also typically manifests in multiple UML diagrams. Figure 4.1 highlights *Feature B* as such an example in orange color. In the feature model, every variable feature may have parent features, child features and may be involved in variability constraints. In the UML model, the functionality that this feature realizes in a software product must be specified precisely and must be linked to the actual feature in the feature model, using a specific variability mapping approach (recall Section 3.2.2). As Figure 4.1 shows, the overall specification of a variable feature, which includes its dependencies to other features and its detailed requirement specification, is distributed over several diagrams that may be instances of different diagram types. Hence, the specification of variable features is inherently scattered (i.e., fragmented) in such state-of-the-art approaches, as highlighted for *Feature B*. This is a fundamental problem that leads to various inefficiencies in the product line engineering process, most importantly the two following ones.

4.2 High Specification and Consistency Maintenance Efforts

heightened efforts

Information scattering of feature specifications over multiple separate diagrams leads to heightened efforts for variability specification and consistency maintenance. Whenever a new feature is created in the variability model, its functional requirements need to be annotated in every other requirements diagram (or similar, depending on the variability mapping approach used). A product line requirements engineer needs to assure that every model element in one of the diagrams is either correctly mapped or correctly *not* mapped to a specific feature. Whenever the variability model changes, the specification of all changed features needs to be adapted in all impacted diagrams. Similarly, whenever the requirements model changes, the variability model and the mappings have to be checked and possibly adapted as well. The manual specification and review of all these mappings is a major effort, especially when the completeness and consistency of all mappings between variable features and the concerned model elements in the requirements model need to be assured. The maintenance and evolution of the detailed specification of variable features is also quite laborious because these state-of-the-art approaches scatter the specification of single features over many different diagrams, recall Figure 4.1. Hence, both creating a complete requirements and variability model and maintaining its consistency during its evolution will require major and unnecessarily high efforts in such a state-of-the-art solution.

4.3 Weak Impact Comprehension of Variability Binding Decisions

scattered feature impact

A potential customer in a software product line context is primarily interested in understanding the software product line as a whole. This understanding helps the customer stakeholder to configure and derive the specific application software product that ideally fits his or her needs. The variability model (e.g., the feature model), which is typically used for product configuration, offers only a quite limited description of a feature's impact on the product's functionality, when it gets selected or deselected. From a requirements point of view, it only shows the feature's name and its dependencies on other features. While this abstract name is enough information in many cases, the detailed specification is also required frequently. When a customer has to select one out of several features, for example, she may need to look into various other UML diagrams to understand the detailed feature specification. Only locating and reading all the specified information for a variable feature allows to make a really informed choice that minimizes the risk for mistakes. Finding this detailed feature specification requires numerous context switches between various diagrams of different types (i.e., in today's modeling tools) and, thus, hampers a straightforward understanding significantly. Furthermore, when additional constraints are involved that also require a selection or deselection of other features for a valid selection

or deselection of a specific feature, this is part of the feature's overall impact as well. When variability constraints are involved, gaining a precise understanding of the real impact of a specific variability binding decision on the actual application product's functionality becomes even more intricate, hence. This necessity to switch back and forth between the variability model and various separate UML diagrams, for any variable feature currently under consideration, hampers the impact comprehension of variability binding decisions considerably. Further, it also hampers any maintenance and evolution activities. The root of this problem is that state-of-the-art solutions for requirements modeling (recall Chapter 2) and variability modeling (recall Chapter 3) are not capable of showing the complete impact of a feature selection or deselection in a single, consistent view. They lack the appropriate concrete syntax for doing so.

split between configuration and generation

Moreover, the fact that state-of-the-art approaches separate the variability model from any other conceptual modeling (i.e., which is called *orthogonal* variability modeling) is possibly also the reason for the existing separation of (i) variability configuration and (ii) product generation in state-of-the-art approaches. Variability is typically configured in a feature model, decision model, or in an OVM model. The product's requirements specification is typically generated by the push of a button after this configuration is completed. The real impact of any taken variability binding decision on the requirements model, thus, is not visible during the configuration. Only some tools like pure-systems GmbH's pure::variants or Voelter's projectional language workbenches [Voelter, 2010] already allow a live filtering of the requirements model already during product configuration, since recently. Nevertheless, these tools still require a user to browse various separate UML diagrams, typically also in a separate tool. Handling the functional specification and the variability model in separation leads to an unnecessarily weak impact comprehension, when evolving the product line requirements model or when deriving new product configurations. This may further cause unnecessary additional costs for context switching during a variability configuration or possible mistakes, which may make the product derivation process even more costly.

Part II

Description of the SPREBA Approach

CHAPTER 5

Basic Idea – Premises for a New Approach

This chapter introduces the basic concepts of a new approach to product line and requirements modeling. This approach builds on a requirements modeling language that has a fully integrated concrete syntax and visual notation and on a compositional approach to variability modeling, on this basis. This completely avoids any information scattering of the specification of variable features. With additional tool support, as introduced in the remainder of Part II, this eventually solves all the problems illustrated in Chapter 4.

The presented approach builds on three fundamental premises that are inherently different from existing state-of-the-art approaches (e.g., the UML and feature modeling). We summarize these three concepts as *Integration*, *Composition*, and *View Generation*—see Section 5.1. This new approach allows handling the commonality and the variable features as primary concerns in the concrete syntax of the used modeling language. To illustrate the application of these new concepts a real-world product line example is outlined and discussed in Section 5.2. We chose the variability of the Audi Q5 sports and utility vehicle's infotainment system as an example, which is intuitive and also offers a significant amount of variability and variability constraints. We also use this example throughout Chapter 6. This chapter focuses on the basic ideas of the presented new approach. Chapter 6 will subsequently continue to introduce the necessary language and notational extensions on a more detailed level.

5.1 Integration, Composition, and View Generation

Existing approaches to requirements modeling for software product lines all lead to problems as depicted in Chapter 4. These problems primarily result from the underlying requirements and design modeling concepts that today's variability modeling approaches build upon. Most importantly, this includes the distribution the requirements model into many separate diagrams, as realized in the UML, recall Section 2.2.1.

prerequisites for SPREBA

The SPREBA approach[1] builds on *different* state-of-the-art work that allows nipping these resulting problems (Chapter 4) already in the bud. SPREBA variability models build on the three principles of *Integration* (i.e., notational integration), *Composition* (i.e., compositional variability specification), and *View Generation* (i.e., the generation of abstract views), which are not present in today's requirements and variability modeling approaches. Figure 5.1 visualizes these key differences. The concepts shown on the left of Figure 5.1 were already introduced in the Chapters 2 and 3. The basic ideas on which the SPREBA approach builds, which are also the prerequisites for realizing SPREBA, are shown on the right of Figure 5.1 and are explained in the following.

Integration

integrated concrete syntax

The SPREBA approach can be applied to any modeling language that has a fully *integrated concrete syntax and visual notation*. Chapter 2 has quite comprehensively discussed the UML and the ADORA approach. While the UML splits the concrete syntax of the model over multiple diagrams of different types, ADORA, on the other hand, does not do so and maintains a fully integrated concrete syntax and visual notation. The top two diagrams of Figure 5.1 highlight this key difference. While a variability modeling solution based on the UML inevitably needs to deal with multiple separate diagrams, this is *not* the case for an ADORA model. An ADORA model is always a single, fully integrated diagram. It visualizes the various facets of structure, behavior, user interaction, context, and more all in a one diagram—see the top-right diagram in Figure 5.1. This is the key difference to state-of-the-art requirements modeling languages that has allowed us to develop a novel kind of variability modeling approach, as follows.

not just ADORA

The work presented in this thesis has been driven by the ADORA approach. However, the SPREBA concepts can essentially be applied to *any* modeling language that has an integrated concrete syntax. Creating an integrated concrete syntax and visual notation for very large languages like the UML would admittedly be a major challenge, however. On the other hand, many smaller languages that have a more specific scope still have an integrated concrete syntax. For example, domain-specific languages (DSLs) are typically defined as integrated notations. For

[1] SPREBA stands for Software Product Line Requirements Engineering Based on Aspects and was a research project funded by the Swiss National Science Foundation from 2008 to 2011 [Glinz, 2008b] [Glinz, 2010a].

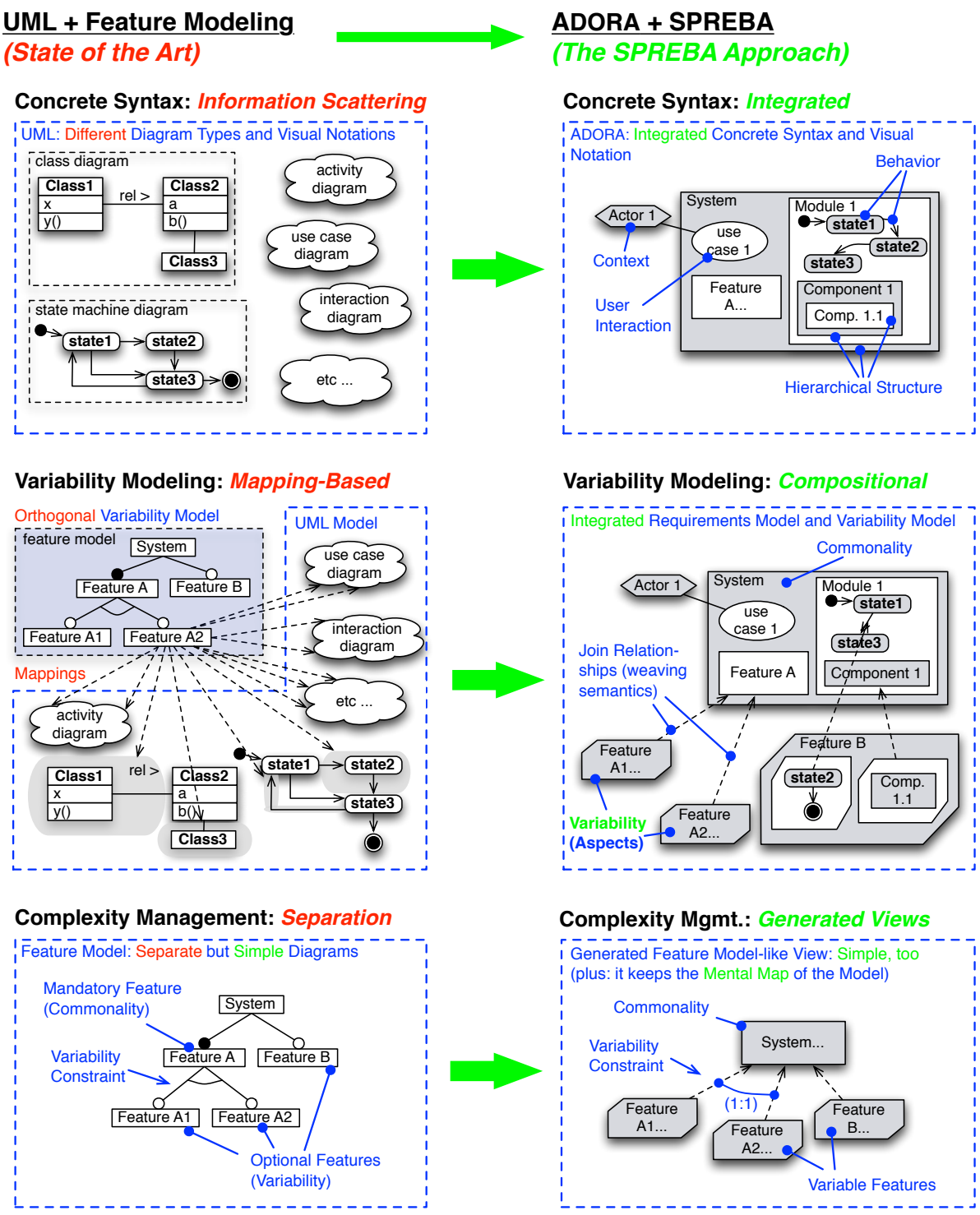


Figure 5.1: An overview of the three major underlying paradigms of our approach: *integration*, *composition*, and *view generation*.

such languages the development of multiple concrete syntaxes to present the *same* information in different forms (e.g., textual, graphical or in a specific data structure) may also make sense [Kleppe, 2008]. In general, it is important to have a concise abstract syntax to allow correctness-preserving model transformations between these representations, when multiple concrete syntaxes are used. Whenever all the specified information can fully be shown in a single language and notation, then this specific concrete syntax is *integrated*. Therefore, languages that have multiple concrete syntaxes may also satisfy this first prerequisite for the SPREBA approach, when SPREBA is applied to one of these concrete syntaxes that is exhaustive and integrated.

no information scattering

This first prerequisite is the basis for solving the first problem identified in state-of-the-art approaches, namely information scattering (Section 4.1). An integrated concrete syntax does by definition *not* distribute information about the model onto separate diagrams. The second prerequisite for SPREBA, which is *Composition*, must already build on a fully integrated concrete syntax.

Composition

no orthogonal variability model

The SPREBA approach uses a compositional approach (e.g., an aspect-oriented approach, see Section 2.4) to model variability, or variable features, respectively. Importantly, *no additional, orthogonal variability model* is used in SPREBA. Instead, the compositional feature model specifies all this information. Existing approaches already applied compositional approaches for variability modeling—recall Section 3.2.2. However, these all build on modeling languages that rely on a scattered concrete syntax of the model. Hence, these approaches still do not prevent the information scattering of variable feature specifications. From a more distant perspective one can also observe a general trend towards more compositional approaches for variability realization mechanisms, see [Bosch and Bosch-Sijtsema, 2010], for example. However, all of these existing compositional approaches still rely on an additional, orthogonal variability modeling at some point, as a basis for the variability configuration. This is also the case for all existing approaches that use compositional variability realization mechanisms. More concretely, most of them still rely on feature modeling and a particular variability mapping approach, recall Sections 3.2.1 and 3.3.

integrated variability

The SPREBA approach requires no additional, orthogonal variability modeling notation in any form (i.e., as introduced in Section 3.2.1). Rather, variable features are directly specified as aspects (or as compositional increments, in general). Figure 5.1 illustrates this key difference in the two diagrams in the middle. State-of-the-art approaches (left) all introduce an additional dedicated diagram for variability modeling and require some kind of mapping approach to establish traceability between variability model and other diagrams (Section 3.2.2). The SPREBA approach (right) does not require any additional notation to model the variability, but utilizes aspect-oriented modeling to modularize these variable features already in the require-

ments model. This leads to a model where only the commonality (i.e., those parts of a model that are required in all products of the product line) is left as a conventional model and all the remaining requirements specification becomes part of a *variable feature*. Such variable features are modularized with aspect-oriented modeling in ADORA's realization of SPREBA, see Figure 5.1. This compositional approach, hence, still keeps the concrete syntax and the visual notation of the model fully integrated, even when variability is introduced.

less variability-related effort

Compared to state-of-the-art approaches, SPREBA leads to less clerical as well as intellectual effort when creating, maintaining, and evolving the variability model—see our empirical results in [Zoller, 2010] [Stoiber and Glinz, 2010a]. Similarly, reviews of feature specifications and of the requirements for a product line as a whole also become less challenging, when appropriate tool support is provided. The decomposition of an integrated requirements model into a model of the commonality and into compositional features makes variable features appear much more as primary concerns, which they actually are in the context of a software product line. Variable features specified with SPREBA include their detailed requirements specifications directly in the same diagram (i.e., via hierarchical nesting of model elements). In the concrete syntax of UML-based requirements specifications in state-of-the-art approaches this rather appears the other way around, since the UML model and its model elements still remain the primary artifacts (i.e., in state-of-the-art UML modeling tools) and variable features are merely annotations. The UML model can also be organized such that variable features become modules. This allows an easier traceability but still leaves the specification scattered over multiple diagrams, however (i.e., the various UML diagrams and the variability modeling). When trying to view the specified variable features in the feature model as primary artifacts, then their detailed specification can not be visualized with any existing language and notation. It can be argued that describing and understanding the specification of variable features in separation, as done in state-of-the-art work (see the middle-left diagram in Figure 5.1), is in general not easy. In the SPREBA approach, these variable features, their variability constraints, and their detailed functional requirements specifications are specified and visualized in the same diagram and visual notation. This allows a significant reduction of the variability-related effort and, hence, solves the second problem of state-of-the-art approaches, as highlighted in Section 4.2.

improved feature impact comprehension

Furthermore, similarly to the reduced effort for feature specification reviews, SPREBA's compositional approach also leads to an improved understandability of the effects of taking a variability binding decision (i.e., when reasoning about selecting or deselecting a variable feature). From a requirements modeling point of view, the model specified with the SPREBA approach does directly show the complete impact (i.e., all additions and/or changes) that a selection or deselection of a variable feature implies.[2] Variability binding decisions also impact other variable features through feature hierarchies and/or constraints. These constraints can exhaustively be visualized in a SPREBA model (Section 6.1). The SPREBA approach includes an advanced

[2]Explanatory note: When a feature is selected it is composed automatically and when it is de-selected it is removed automatically, in SPREBA—see Chapters 7 and 8.

automated analysis solution of variability constraints that calculates the complete set of constraint propagations for any potential manual change of a variability binding decision (i.e., such that all constraints stay satisfied when a variable feature is selected, deselected or changed to its complement value, or back to undecided, see Sections 8 and 9). In existing state-of-the-art approaches, the impact of a variable features is more cumbersome to comprehend for engineers because it is scattered over multiple diagrams and a similar preview on the constraint propagations is often not provided. Therefore, SPREBA also significantly mitigates the problem of weak impact comprehension of variability binding decisions in existing state-of-the-art approaches, as described in Section 4.3.

View Generation

catching up with scalability

While *Integration* and *Composition* can provide substantial advantages, as they allow solving the problems described in Chapter 4, they also cause scalability issues. A single integrated diagram quickly grows much larger than any diagram created within a UML model and, thus, can quickly lead to a visual and cognitive overload. To deal with such large diagrams, *View Generation* is an essential concept that has been explored in the context of ADORA since [Berner et al., 1998a], recall Section 2.3.3. Several view generation mechanisms have been developed to improve the visual scalability of ADORA models and to help controlling and mastering the visual complexity of a large diagrams. These include fisheye-zooming (i.e., vertical abstraction), filtering of model elements of particular types (i.e., horizontal abstraction), human-friendly automatic line-routing and automatic label placement. Reinhard has provided a general and recent overview on state-of-the-art visualization techniques for visualizing hierarchical models (i.e., ADORA models) [Reinhard, 2010]. Further, in [Stoiber et al., 2008] we already demonstrated that these concepts can straightforwardly be used with ADORA and SPREBA as well.

dynamically generated views

The bottom-right diagram in Figure 5.1 shows a dynamically generated abstract view which we call the *feature diagram-like* view. This view is particularly valuable for detailed product line requirements models. The feature modeling notation is a quite powerful concept in state-of-the-art approaches. Feature models provide a comprehensive overview of all product features (which can be considered high-level requirements) and also include the complete variability model and all constraints. In general, the visual syntax of a feature modeling (Section 3.2.1) is typically not regarded as overloaded (i.e., from a practical point of view) and feature models typically do not grow all too large (i.e., there are exceptions, of course) because features are still a quite abstract concept.

A SPREBA model, on the other hand, with all functional requirements specification details visualized, is much less a general and abstract overview. With all information visualized, a SPREBA model quickly becomes too complex to comprehend for human engineers. However, when building on Reinhard's existing complexity management techniques for graphical models, we can easily create a macro-function that collapses all feature nodes and all commonalities, and

visualizes all variability constraints (as follows in Section 6.1.2) [Reinhard, 2010]. The result of such a macro-function will be a dynamically generated view that resembles an abstraction level very similar to a classic feature diagram. The precise similarities and differences will be investigated more closely in the upcoming Sections 5.2, 6.2, and 12.4. The bottom-right diagram in Figure 5.1 visualizes such a *feature diagram-like* view as a dynamically generated view on a comprehensive ADORA and SPREBA model. It also visualizes the xor variability constraint between the variable features $A1$ and $A2$, which is also visible in the classic feature diagram on the left-hand side. Feature A is not visualized, though, because it is a mandatory feature of the root feature and, hence, part of any product. Therefore, this feature remains part of the commonality and is not modularized as a variable feature, as it is in fact not variable.

preserving the mental map

An important property of ADORA's view generation capabilities, which are also required for SPREBA, is the preservation of the mental map of the model for any generated view [Reinhard et al., 2008]. The bottom-right diagram in Figure 5.1, for example, still presents the same relative alignment of the commonality and the variable features as the middle-right diagram does, which presented a more detailed view. Such a preservation of the mental map helps engineers to re-orient themselves better and faster whenever the currently shown view (i.e., generated view) is changed. This minimizes the required human efforts for re-comprehending any newly generated view.

View generation is not a contribution of SPREBA per se, but rather a way to deal with the increased visual complexity and the size of the model at hand in a smart way. The goal of view generation is to dynamically adjust the level of abstraction of the visualized model, to generate views on a SPREBA model that are about as simple as UML diagrams or feature diagrams are. Using view generation to *navigate* a SPREBA model and to maintain the mental map of the model, as presented in [Reinhard et al., 2008], can further improve the understandability of the model. On the other hand, the current ADORA tool implementation still is not mature enough to provide the necessary human-friendliness, performance and ease of use that is be required to make this concept really a clear advantage. SPREBA itself bears very considerable advantages over state-of-the-art approaches, as recent empirical experience has shown (Section 14, for example). A stable and seamlessly integrated realization of powerful view generation capabilities is crucial for reaching such results, though. ADORA's still quite prototypical tool implementation has actually dampened many of these otherwise persistently positive results, however. A user-friendly, reliable, and powerful implementation of view generation, hence, is indispensable for the success of SPREBA.

ADORA has these premises

The ADORA approach (i.e., ADORA is both a language and tool) realizes all three of these preliminaries for SPREBA: *Integration*, *Composition*, and *View Generation*. In fact, the SPREBA approach has been developed on basis of the ADORA approach since [Stoiber et al., 2007]. SPREBA is not limited to ADORA, however, and could be realized with *any* modeling language that has an integrated concrete syntax and notation and powerful composition and view generation capabilities. For example, even for the UML one could come up with a new and integrated

concrete syntax and visual notation, a compositional approach suitable for this new integrated concrete syntax and new, powerful view generation concepts. This may be rather challenging and pose substantial new research and engineering challenges, but we believe that this may actually be feasible. For the remainder of Part II of this thesis, however, ADORA will serve as a vehicle for introducing and illustrating the SPREBA concepts.

comparison with AHEAD

Batory et al. have previously introduced the Algebraic Hierarchical Equations for Application Design (AHEAD) approach [Batory et al., 2004]. AHEAD shows how programs and non-code representations can efficiently be handled and synthesized with feature refinements, when deriving products from a product line. While SPREBA only focuses on requirements and variability modeling in this thesis, AHEAD also deals with different types of software artifacts over the whole software lifecycle. These include equational specifications, Java, and non-Java artifacts, for example. The work on SPREBA and on AHEAD does not overlap, in our opinion, but rather complements each other. In a sense, the here presented approach could also be used as a graphical representation for a set of AHEAD modules. This would allow using the solid fundamental concepts of AHEAD as existing work. However, merging the two would also require a notational integration of the concrete syntax, to realize SPREBA's first prerequisite of *Integration*. This is out of scope of this thesis, though, but future research in this area may be very worthwhile.

5.2 A Real-World Example

Audi's online car configurator

To illustrate the SPREBA variability modeling approach we briefly show how these concepts can be applied on a real-world example. We chose a realistic example from the automotive industry, namely the infotainment system of the Q5 sports and utility vehicle (SUV) from Audi AG. We derived our example data directly from Audi's online car configurator.[3] This example is particularly well suited for three reasons: (i) it is a software-intensive system, (ii) it has about the right amount of variable features to be illustratable within this thesis, and (iii) it contains several variability constraints.

The selectable equipment in Audi's Q5 configurator was typically not hierarchically ordered and not cleanly decomposed into single features. For example, one could select quite compound options like a *'3-spoke multifunction sports steering wheel'* or a *'4-spoke multifunction steering wheel'* and not separate single features like, e.g., *3-spoke*, *4-spoke*, *sports edition*, or *multifunctional*. Many additional features in Audi's configurator also required other options to be selected, which yields variability constraints. This section focuses only on showing how the requirements and variability of the Q5 infotainment system can be modeled in an *integrated* and *compositional* way, to demonstrate SPREBA's basic feasibility. It also shows how *view*

[3] See http://konfigurator.audi.de/ (checked on June 30, 2012).

generation can be applied on this basis. This section does not yet provide detailed language definitions for SPREBA, however, which follows later in Chapter 6 and Section 9.1.

Audi Q5 infotainment system

Every Audi Q5 has a basic infotainment system installed as standard (e.g., a basic board computer and radio system). There are plenty of optional upgrades. For example, a prospective buyer can upgrade the *Basic Radio System* (that is standard equipment) to an *Advanced Radio System* or to a *Navigation System*.[4] We assume that advanced radio systems are developed separately from the navigation system and therefore have separate specifications, despite potential overlaps in functionality. We also assume that both extend the basic radio system's functionality (we are not aware of the actual software architecture used within the company). Every prospective buyer can choose whether he wants to upgrade to an advanced radio system or to a navigation system, which are mutually exclusive options—the specification of variability constraints will follow in Chapter 6. Further, *Bluetooth Hands-free for Mobile Phone* is also an optional feature, which can extend an advanced radio system or a navigation system to work as a handsfree set and to wirelessly stream music from the mobile phone. If a prospective buyer chooses a navigation system and the Bluetooth hands-free set, then he can additionally add a *Mobile Phone Docking Station with Voice Control* as an extension of these two. This adds a charging capability for the mobile phone, allows the use of the car's internal 3G antenna for improved connectivity and allows browsing the phone's contacts over the navigation system interface. The *Rear Seats Entertainment* and the *TV* are further optional features. The *Rear Seats Entertainment* feature is purely optional. The *TV* feature can only be realized when the *Navigation System* is also selected. Further, the *TV* feature allows receiving terrestrial TV programs on the navigation system's screen. When both the *TV* and the *Rear Seats Entertainment* are selected, then the TV reception will also be available on the rear seat monitors. However, the TV feature does not require the rear seats entertainment to be selected as well.

SPREBA model

Figure 5.2 shows an example SPREBA requirements and variability model of this above described infotainment system. The model shows a very abstract specification of the commonality and eight variable features. For seven of them the actual requirements specification is not shown, for space reasons. For the feature *Mobile Phone Docking Station with Voice Control* a simple behavioral specification of the charging functionality is shown in ADORA notation. This specification is directly modeled in the aspect container that specifies this variable feature, see Figure 5.2. When this feature is composed, the *charging* specification as well as the specification of the *use internal 3G antenna* component will be composed into the *Infotainment* component in the commonality (recall ADORA's aspect weaving semantics, see Section 2.4.2). The functionalities *browse phone contacts* and *voice control* could also be separate, stand-alone features. However, Audi's online configurator included both of these functionalities as part of the *Mobile Phone Docking Station with Voice Control* feature, which implies that they always vary together and do not require separate single features. Thus, both of them are modeled as

[4] Note that trademarked brand names, like, e.g., *Radio Chorus*, *Concert* or *Symphony*, etc., were generalized to focus on the underlying concepts and not advertisement.

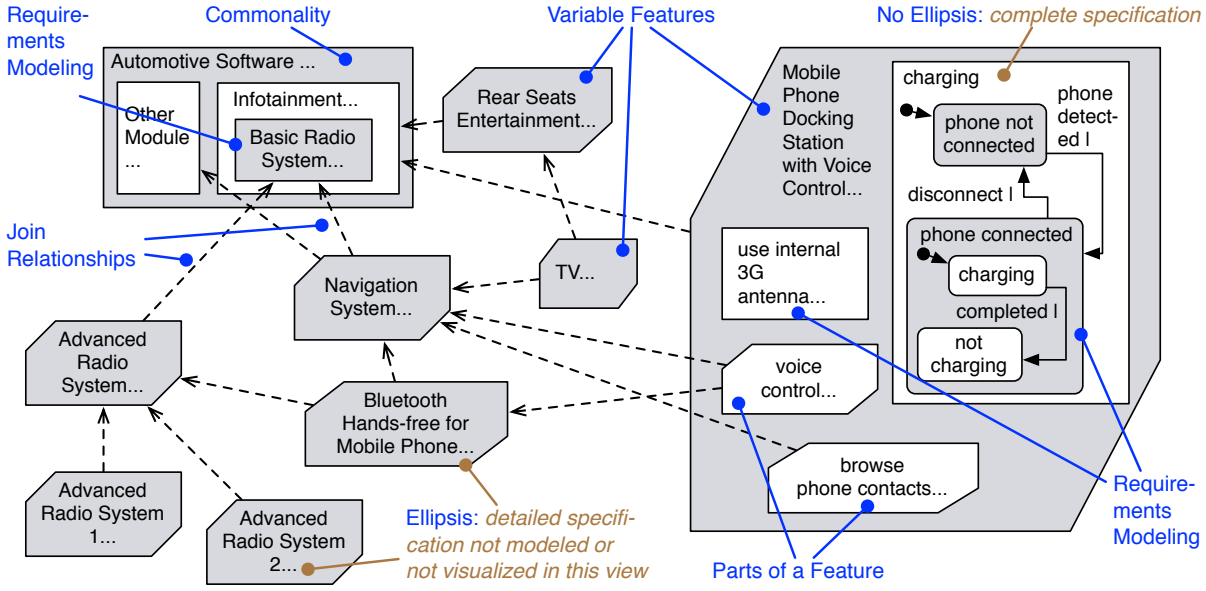


Figure 5.2: An example SPREBA requirements and variability model of Audi's Q5 infotainment system (not showing any variability details and constraints).

parts of a feature and with join relationships (i.e., composition directives) to extend the navigation system's functionality. The detailed specifications of these two features are not visualized in the view shown in Figure 5.2, which is indicated by the ellipsis (i.e., the three dots) behind the respective model element's names. As shown for the feature *Mobile Phone Docking Station with Voice Control*, the detailed requirements specification of all other features and of the commonality is directly nested within these more abstract model elements in the SPREBA product line model.

features as first-class entities

The SPREBA variability model, as depicted in Figure 5.2, differs from the variability described in Audi's online car configurator (which is remotely similar to wizard-based or questionnaire-based approaches) and from feature models or other state-of-the-art variability models (recall Section 3.2). A SPREBA model essentially covers the full requirements specification of the system at hand and introduces a modularization of variable features at a level of abstraction that is similar to that of software components or features (e.g., as in a feature diagram). While the model still represents a full system specification, features are now modeled as first-class entities already within the requirements modeling language and notation. As Figure 5.2 shows, features and the association of requirements modeling to specific features is clearly visible in a SPREBA model. Additional textual descriptions, as they frequently appeared in Audi's configurator, can easily be added—either as attached comments to the respective model elements, or as textual specifications in a possible future version of ADORA [Glinz, 2010b]. To maintain the benefits

of SPREBA and not to get users into trouble with increased visual complexity, when the model grows, powerful support for view generation is critical for success.

hierarchies by functional decomposition

Features are modeled as functional increments, in the SPREBA approach, while in Audi's original configurator they are grouped into themes (e.g., interior, exterior, seats, infotainment). Functional dependencies, thus, are not directly visible, but needed to be added as dedicated constraints in Audi's configurator (e.g., the Bluetooth interface is only available in combination with an advanced radio or a navigation system). In a SPREBA model, functional dependencies are directly visible through the defined join relationships (see the dashed arrows in Figure 5.2). Thus, depending on the detailedness of the model, many constraints that derive from functional dependencies are inherently defined in a SPREBA variability model and do not need to be elicited and maintained manually. For example, the *TV* feature extends the functionality of the *Navigation System* and the *Rear Seats Entertainment* features. This already specifies a non-trivial hierarchy constraint, resulting from a heterogeneously cross-cutting variable functionality (see Chapter 10 for a classification of cross-cutting features). Such a cross-cutting can not be modeled in classic feature diagrams—a more detailed narration follows in Chapter 6. While Czarnecki et al. already introduced the concept of aspectual features [Czarnecki et al., 2005b], constraints resulting from cross-cutting features still need to be defined and maintained manually in such state-of-the-art approaches. In SPREBA, such dependencies are directly defined and visible through the topology of variable features and join relationships.

differences to feature modeling

In summary, the main differences between a SPREBA variability model and a feature model are the following. A SPREBA model *contains* all the detailed requirements specification and does not require traceability to other requirements artifacts, while a feature model does. A SPREBA model is the result of a direct decomposition of the requirements model into a commonality (i.e., or multiple commonalities) and variable features. A feature model, on the other hand, is not always developed conjointly with the functional requirements specification. This aspect-oriented decomposition of the SPREBA model implies that dependencies between requirements are reflected more naturally than in a feature diagram. Depending on the detailedness and expressiveness of the underlying requirements modeling language, additional constraints may still be necessary, however. These can easily be added on top of such a feature-oriented specification, as follows in Chapter 6, similarly to how constraints are specified in feature models. Like other decision-based approaches, a SPREBA variability model also only focuses on the *variable* parts of the model and considers the rest as commonality. Hence, it only modularizes the *variable* features and does not require to specify any mandatory features with dedicated aspects. Feature diagrams, on the other hand, do typically also include mandatory features. In a SPREBA model, these are either part of the commonality or of those features they are a sub-feature of. Feature models can also be "minimized" in a similar way, however, as elegantly shown in [Czarnecki and Wasowski, 2007], for example. Otherwise, when no requirements details are specified, or visualized in the currently generated view (we call the latter a 'feature diagram-like' view), a SPREBA model still bears a considerable resemblance to a feature diagram.

CHAPTER 6

Language Concepts – A Novel Boolean Decision Modeling Concept

This chapter introduces a new variability modeling approach that is parsimonious and has particularly been designed for integrated and compositional variability modeling. The chapter presents a complete overview of the required extensions to a language's concrete syntax and notation that need to be introduced for SPREBA variability modeling. The prerequisites for applying this variability modeling approach are the use of an integrated visual syntax and a compositional approach (Chapter 5). This chapter uses the ADORA language (Section 2.3) and ADORA's aspect-oriented modeling capabilities (Section 2.4.2) as an example approach that satisfies these prerequisites. The presented variability modeling concepts, as introduced in this chapter, however, are generic and can be applied to extend any language and notation that satisfies the premises illustrated in Chapter 5.

variability on the level of join relationships

Existing variability modeling approaches handle variability with orthogonal variability models and mapping-based approaches. The most widely used variability modeling notation is feature modeling. Feature models differ considerably from SPREBA models, however. In SPREBA the handling of variability is different, as a compositional approach is used for variability modeling. The configuration is not based on selections and deselections of features directly, but rather on so-called *decision items*, which are associated with the join relationships (i.e., the weaving semantics) of variable features. Join relationships define where-to and how the requirements of a variable feature needs to be woven, if selected. Otherwise, no weaving will be performed. This chapter shows how aspects can be utilized and extended to precisely express variable features

in such a way. This approach could also be sketched as making decisions by directly configuring variation points. Furthermore, allowing the modeling of cross-cutting variable features also requires additional, sophisticated concepts for distinguishing between weak and strong hierarchical dependencies between variable features. In feature diagrams, only strong hierarchical dependencies exist. SPREBA also requires weak hierarchical dependencies, however, as motivated and described in this chapter. In summary, dealing with variability in an integrated notation and on a compositional level together with allowing cross-cutting variable features and weak hierarchical dependencies in the variability model are the major novelties of SPREBA's Boolean decision modeling concept.

origins of SPREBA

The original idea of using a dedicated decision modeling concept for aspect-based variability modeling has been presented in [Stoiber et al., 2007] and was further discussed in [Glinz, 2008b]. This original idea was essentially based on Schmid and John's table-based decision modeling notation and variability management approach [Schmid and John, 2004]. This classic decision modeling concept, as already shown in Figure 3.5, however, made it hard to efficiently specify and analyze variability constraints. Such constraints had to be specified only in part and separately for every decision item in the decision model (see, e.g., [Stoiber et al., 2007, Table 1]). This made the variability specification and documentation eminently difficult. It also made configuration tasks cumbersome, as the overview on these various dependencies was easily lost. Therefore, we decided to develop a new Boolean decision modeling concept that handles both the variability specification for SPREBA models and of variability constraints as good as possible. The result was a new variability modeling concept, called Boolean decision modeling, that was first motivated in [Glinz, 2008b] and in extracts also already presented in [Stoiber et al., 2008], [Stoiber and Glinz, 2009] and [Stoiber and Glinz, 2010b]. This chapter, however, is the first complete and comprehensive presentation of SPREBA's novel Boolean decision modeling solution. This solution integrates much more seamlessly into a SPREBA requirements and variability model (recall Chapter 5) than any existing solution. Section 6.1 illustrates this variability modeling approach with the automotive running example already introduced in Section 5.2. Section 6.2 further provides a short discussion and reflection on these new concepts.

6.1 Boolean Decision and Constraints Modeling

Variability is modeled upon join relationships in ADORA. In the following we first introduce how features, variable join relationships, and decision items connect. A more formal definition of these model elements is presented in Section 9.1. Then, we show how additional constraints are defined and visualized. Finally, the complete *Decision Table* view is introduced, which is used for configuring the variability and which already includes some automated analysis results to improve the understanding of hierarchies, cross-cutting, and the additional constraints between decision items.

6.1.1 Join Relationships, Features, and Decision Items

variable JRs

Join relationships (JRs) are used to define where and how an aspect needs to be composed in an ADORA model, recall Section 2.4.2. Removing all join relationships of an aspect makes the aspect container and all its content semantically void, with respect to the remaining model. Executing all join relationships of an aspect performs a composition, which is called *weaving* in aspect-oriented approaches. This mechanism is required for realizing *variability*. Therefore, variability in a SPREBA model is defined upon join relationships, since this makes it straightforward and natural to either compose or remove variable model elements. Later-on, Chapter 7 will show how a conventional static aspect weaver (i.e., a classic aspect weaver) needs to be refined to allow what we initially called dynamic weaving. Such a dynamic weaver allows to only weave specific join relationships at a time, but not all of them, as typically realized by classic aspect weavers. When used for variability, we call this concept *feature weaving*. Chapter 8 further shows how feature weaving is used to realize stepwise, incremental product derivation. In order to efficiently specify the variability on basis of join relationships, which provides the language and notational basics for all upcoming chapters, we first introduce the concepts of *features* and Boolean *decision items*.

features

Aspects in ADORA have a Boolean attribute *variability*, which defines an aspect to be a 'variable feature' or a 'part of a feature', when set $true$. Otherwise, when the variability attribute is set $false$, the aspect container specifies just a conventional aspect (e.g., a cross-cutting concern). A *feature* in ADORA-SPREBA is an aspect container (AC) that contains variable model elements and that is not hierarchically contained in any other variable aspect container. In the latter case such a nested variable AC would be a *part of a feature*. A feature always has at least one decision item associated with its outgoing join relationships, which steers whether the feature will be composed or removed. Thus, the variable model elements in a SPREBA model are encapsulated into features (which are variable aspect containers) and the actual variability is defined and handled on basis of the feature's outgoing join relationships, with so-called decision items.

decision items

Decision items (DIs) are three-valued "Boolean" variables (Booleans that can also have a third value for $undecided$) that are used to efficiently define the variability and constraints of a SPREBA model. Decision items are associated with join relationships. By default, a decision item has the value $undecided$, but can be set $true$ or $false$ when specializing a product line or configuring a product. Inspired by Schmid and John's work [Schmid and John, 2004], which was in turn inspired by Campbell et al.'s work [Campbell et al., 1990], we document the *decision model* in a table-based format. However, compared to Schmid and John's approach, our decision modeling notation is much more parsimonious—we *only* allow Boolean decision items, similarly to how features are mere Boolean entities in feature models and variants are in OVM, recall Section 3.2.1. In ADORA, the description and handling of decision items is fully integrated into its abstract syntax. Since decision items typically occur on multiple join

relationships, they are handled in a dedicated table-based view. This is to avoid redundancy and to keep the visual complexity of the graphic model reasonably low. Note that the specification of decision items is still fully integrated with the SPREBA model and that the main purpose of this additional view is the centralized representation of the detailed information of decision items. In future versions of the ADORA tool this could also be handled directly in the graphic SPREBA model.

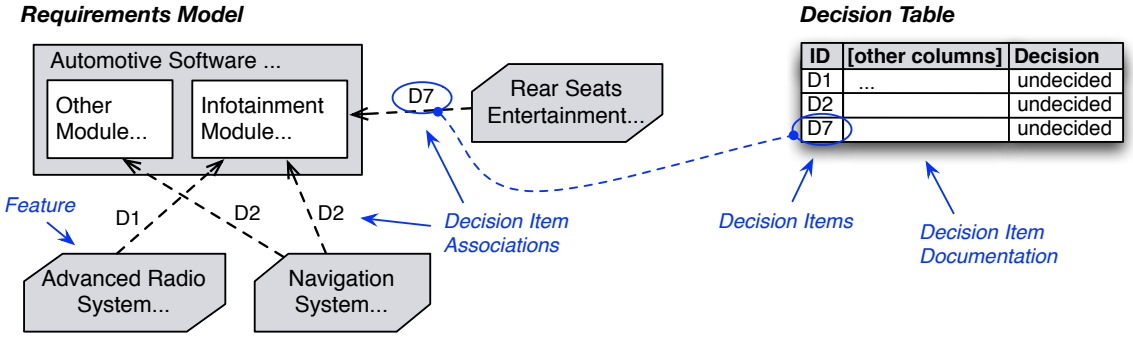


Figure 6.1: An excerpt of our automotive example that highlights how *decision items* are linked with variable features in the graphic model.

modeling variability

Figure 6.1 presents an excerpt of the automotive running example and illustrates how decision items are documented and associated with join relationships of variable features in a SPREBA model. In order to reference a decision item efficiently, a unique decision identifier is introduced, the so-called decision ID. The format 'Dx' was chosen to reference decision items, where x is a unique natural number, starting from 1 and incrementing. Figure 6.1, for example, highlights the association of decision item D7 with the feature *Rear Seats Entertainment*. The presented example is an excerpt of the complete variability model of the Audi Q5 infotainment system, as follows in Figure 6.9. Thus, the decision items are not strictly sequentially numbered, but are instead consistent with all upcoming views. For all three decision items presented in Figure 6.1 there is a one-to-one correlation between the decision item and a particular feature. This is desirable and also the default for variability modeling—a more fine-grained configurability is possible, however, as follows. The *Navigation System* feature provides an example of the reason why we specify the details of decision is a separate, table-based view. This is because decision items are typically associated with multiple join relationships in realistic models. The navigation system feature, for example, impacts the *Infotainment Module* and also another module in the commonality. Therefore, it may contain two parts of a feature and at least two join relationships (assuming that this feature is heterogeneously cross-cutting, see Chapter 10). There can also be multiple JRs on a more fine-grained level that are visualized as one abstract JR in ADORA. For example, when an engineer would collapse the component *Automotive Software* in the currently displayed view shown in Figure 6.1, then the two join rela-

tionships (JRs) associated with decision item D2 would only be displayed as one single abstract join relationship in this new view (recall Figure 2.11 and see [Meier, 2009]).

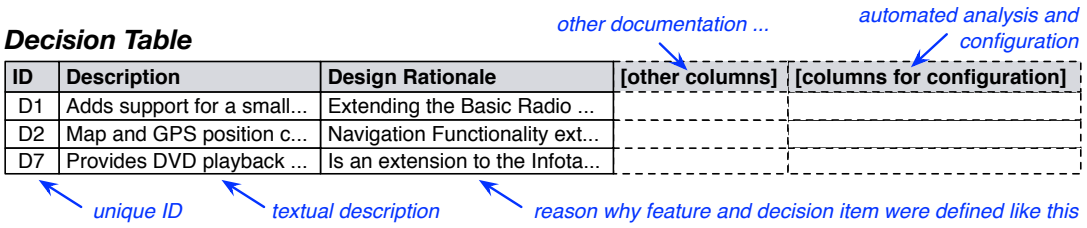


Figure 6.2: An excerpt of the decision table view of our automotive example that shows how additional attributes are centrally documented in a table-based format.

documenting decision items

When creating the variability model for an evolving or planned software product line this often impacts a large number of stakeholders, which in addition are often also distributed globally. In such a constellation many engineers will model, review and change the variability model or the definition and scope of the single variable features, respectively. Whenever an engineer changes the variability model it is, thus, important that a documented design rationale has been captured before, which describes the original intent and reason for this variability. Changes should only be performed in awareness of the originally documented design rationale. Dutoit et al. provide a general overview on recent research on design rationale [Dutoit et al., 2006]. Further, in [Stoiber and Glinz, 2009] we already argued that documenting design rationales for variability in a software product line is crucial, in particular for maintaining and evolving large-scale, multi-stakeholder, and distributed software product lines. Figure 6.2 provides an initial overview on SPREBA's so-called the decision table view. Various attributes of decision items (i.e., every row in the decision table specifies a decision item) are documented in addition to the graphical requirements model, for every decision item. For example, a dedicated design rationale should be provided for every decision item. A textual description of the decision item could also be beneficial to maintain. Many other attributes like estimated costs of realizing this variability, estimated development time, the development sites or partner companies responsible for this variable functionality, names of the responsible engineers, etc., may also make sense and can easily be added to such a table-based format. Figure 6.2 yet only shows a general view on how variability details are documented in the decision table in SPREBA.

hierarchy constraints

Features that extend or change the functionality of other features also induce a strict hierarchy constraint: the selection of a particular feature (i.e., selecting its associated decision item with $true$) requires all other impacted features and parts of features to also be selected with $true$, when all decision items are *strongly* associated with the variable join relationships. Such strong hierarchy constraints are state of the art and used in all current feature modeling languages (Section 3.3.1). They are the classic requires or child-parent dependencies, as already illustrated

in Figure 3.10. These occur equally in SPREBA variability models, when a feature impacts another feature. For example, the *Advanced Radio System 1* or the *Advanced Radio System 2* features in Figure 5.2 strongly depend on the *Advanced Radio System* feature. Whenever the basic *Advanced Radio System* feature is deselected, then there is no possibility for its sub-features' functionality to become part of the product. Thus, they must also be deselected to ensure consistency between the configuration and the actual product model. The other way around, when a sub-feature is selected, all its parent features that it strongly depends upon must also be selected, such that the feature can truly become part of the application product.

when strong dependency becomes inaccurate...

However, when using a SPREBA model, features are often cross-cutting. In SPREBA features are directly modularized with a compositional approach, on basis of an existing functional requirements specification. Hence, they may have arbitrarily many parent features, depending on the chosen decomposition by the involved domain experts. In such cases strong dependencies towards all parent features often become inaccurate, or too restrictive. Consider for example the *TV* feature, as modeled in Figure 5.2. The *Navigation System* and the *Rear Seats Entertainment* features are both optional for an Audi Q5 automobile. The *TV* feature extends the functionality of both of them, but strongly requires only the *Navigation System* feature to be installed. Thus, the *TV* feature is also fully functional when the *Rear Seats Entertainment* feature is *not* realized, as long as the *Navigation System* is also selected. This constellation requires a weak association of the *TV* feature's decision item to the feature's outgoing join relationship(s) that impact(s) the *Rear Seats Entertainment* feature. This is called a weak hierarchical dependency between decision items and is modeled as follows.

weak hierarchical dependency

In general, a decision item must always be deselected (i.e., set $false$) when its specification does not impact the final product specification in any way. This is a general rule, since it does not make any sense to list a decision item (i.e., a variable feature) as selected for any derived product when it did *not* impact the product's functional specification at all. On the other hand, when a decision item is selected, all other decision items that this decision item strongly depends upon must also be selected. However, all decision items on which this decision item only weakly depends upon are not necessarily required. When all dependencies of a decision item are weak dependencies, however, then *at least one* of the targeted model elements must be satisfiable and eventually impact the commonality of the model, to make sure that this decision item really impacts the actual product. Otherwise, the feature does in fact not become part of the product and must be set $false$, such that the configuration is fully consistent with the actual product model. How these dependencies are formally resolved in a SPREBA model will be presented in Section 9.3. In SPREBA's visual notation a strong association of a decision item Dx to a variable join relationship is modeled and visualized by a plain annotation of Dx, while a weak association, that specifies only a weak dependency, is visualized by a bracketed association (Dx) of the decision item to the respective JR.

an example

Figure 6.3 eventually shows how the just discussed *TV* feature's weak hierarchical dependency is modeled with a weak association of its associated decision item. The semantics of the *TV*

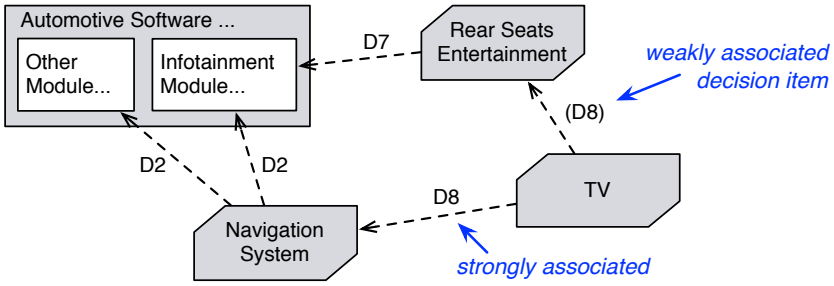


Figure 6.3: A weakly associated decision item *(D8)* illustrated on an excerpt of our automotive example.

feature, which is associated with decision item D8, is that it always requires the navigation system (i.e., D2) to be selected, but does not necessarily require D7 to be selected as well, see Figure 6.3. However, when D8 and also D7 are selected, then the rear seats entertainment functionality will be extended and also include the *TV* feature's functionality.

6.1.2 Additional Variability Constraints and their Visualization

Hierarchies and cross-cutting with strong and weak dependencies can already specify a significant amount of dependencies between decision items in a SPREBA variability model. However, they can not express all constraints that typically occur in a software product line. For example, constraints that originate from a business context or a marketing strategy have to be added in addition, as they are no functional or technical dependencies. Such further constraints are typically always of one of the two following types: (i) cardinality-based constraints, that may occur at variation points where multiple variable features could be woven and (ii) arbitrary variability constraints that could result from e.g. data dependencies that are not specified in the requirements model. We call the first type of constraint *variation point constraint*, which is similar to what others called cardinality-based constraint of a group feature [Czarnecki et al., 2005b], node types [Schobbens et al., 2006] or a relationship among features [Benavides et al., 2010]. We call the second type *arbitrary Boolean logic constraint*, which is similar to what others called arbitrary propositional constraint [Batory, 2005], additional constraint [Czarnecki et al., 2005b] [Batory, 2005], or cross-tree constraint [Benavides et al., 2010]. The reason for using these two different types of constraints is as follows.

intricacy of Boolean algebra

Pure Boolean algebra would actually suffice to express any arbitrarily complex constraint that may occur in a real-world software product line. However, Boolean algebra is very cumbersome to use for particular types of constraints that actually occur quite frequently, like alternative constraints among sets of features, for example (i.e., a logical *xor* dependency). If the features

identified with the decision items D1, D2, and D3, for example, form such a mutual exclusion (i.e., alternative features of which exactly one must be selected), this would yield a ridiculously long formula in pure Boolean algebra: $(\mathrm{D1} \wedge \neg\mathrm{D2} \wedge \neg\mathrm{D3}) \vee (\neg\mathrm{D1} \wedge \mathrm{D2} \wedge \neg\mathrm{D3}) \vee (\neg\mathrm{D1} \wedge \neg\mathrm{D2} \wedge \mathrm{D3})$. The reason for such a long formula is that Boolean operators only allow *binary* operators that can not express dependencies among more than two Boolean variables at a time. For defining an xor constraint among four or more decision items, the formula would even be much longer. For other constraints, however, like requires dependencies, Boolean logic suits very well, though. For example, the simple constraint that feature D1 requires feature D2 can straightforwardly be specified as $\mathrm{D1} \Rightarrow \mathrm{D2}$. For this reason, we split the definition of constraints into two parts: cardinality-based constraints (i.e., *variation point constraints*), to specify all sorts of group-based constraints, and Boolean algebra-based constraints (i.e., *arbitrary Boolean logic constraints*), for all the remaining, additional dependencies.

variation point (VP) constraints

Variation point constraints (VP constraints, in short) are specified by referring to the set of decision items that are involved in the constraint and specifying the minimum and maximum number of these decision items that need to be set *true*. We chose to call this type of constraint a *variation point constraint* because it is typically defined among multiple decision items (i.e., variable features) that compete at the same or a closely related variation point (i.e., join point) in the model. In general, a specific location within an artifact at which a modification must occur is typically called a *variation point*, today, both in the literature and in industry. In SPREBA, such variation points are the join points, defined by variable join relationships. *Variation point constraints* (or *VP constraints*) in SPREBA further are cardinality-based variability constraints for closely related, competing variants at a particular variation point.

Our notion of cardinality-based VP constraints is quite similar to what Batory has defined as the *choose* function [Batory, 2005]: $choose_{n,m}(e_1...e_k)$ means that at least n and at most m of the expressions $e_1...e_k$ are true, where $0 \leq n \leq m \leq k$. Also Czarnecki et al. provided a very similar definition [Czarnecki et al., 2005b]. They additionally introduce so-called *group features*, which are additional features that may not imply any functionality themselves, but are used for specifying cardinality-based constraints. Such group features are not modeled in SPREBA, however, because they typically do not directly refer to any functionality or requirements. Schobbens et al.'s concept of *node types* is similar to such group features as well [Schobbens et al., 2006].

In SPREBA every variation point constraint is defined as a dedicated data structure and referenced with a unique identifier VPx, where x is a natural number starting at 1 and incrementing, analogously to how decision items are referenced. The documentation of a textual design rationale and further important information is also encouraged. Such information may be invaluable in future maintenance tasks.

VP examples

Figure 6.4 shows an example variation points table view with two variation point constraints specified as they occur in our automotive running example, recall Figure 5.2. These two cardinality-based VP constraints are directly derived from Audi's Q5 online car configurator.

Variation Points Table

ID	Design Rationale	minCard	maxCard	DecisionsInvolved
VP1	Both the Advanced Radio System (ARS) and Navigation System can not be chosen. But both of them can be deselected.	0	1	D1, D2
VP2	Either ARS 1 or ARS 2 has to be chosen.	1	1	D3, D4

Figure 6.4: A variation points table view that specifies two cardinality-based variability dependencies of our automotive running example.

For Audi's Q5 infotainment system one can always either choose an advanced radio system (i.e., D1 = $true$) or a navigation system (i.e., D2 = $true$). It is not allowed to choose both. It is allowed, however, to choose none of the two and to stick with the basic radio system functionality that comes as standard equipment. This constraint, thus, neither is a logical OR nor a logical XOR dependency, but must be specified with these concrete cardinalities. Based on the two involved decision items D1 and D2 the required minimum cardinality is 0 and the required maximum cardinality is 1. A dedicated textual design rationale should be added as well, for this particular VP constraint. Figure 6.4 shows a variation points table view where all this information is specified as VP1. Further, a second VP constraint in this automotive example is that whenever an advanced radio system (ARS) is chosen, also one particular ARS out of two possibilities ARS 1 (D3) and ARS 2 (D4) has to be chosen. It is not allowed to de-select both when the basic ARS feature is selected (i.e., D1 = $true$) and it is also not allowed to select both, in any case. It is allowed that both of them are deselected, however, but only when the basic ARS feature (i.e., their parent feature) is deselected, too. This is specified by a variation point constraint with the cardinalities $1{:}1$ (i.e., for the *min* and *max* number of decision items that must be set $true$) over the decision items D3 and D4—see VP2 in Figure 6.4. Similarly to how this is handled with feature models, the constraint VP2 is only relevant when the basic ARS feature (i.e., D1) is either $selected$ or set $undecided$. Otherwise, when the parent feature D1 is set $false$, all child features (e.g., D3 and D4) that strongly depend on this feature must also be set $false$ and the constraint VP2 does not need to hold. A more precise and complete formal semantics of how variation point constraints are formally interpreted is provided later-on in the Sections 9.1 and 9.2.

VP visualization

VP constraints can be visualized trivially among the variable JRs they are defined upon in the graphic SPREBA requirements model. Figure 6.5 demonstrates how the VP constraints VP1 and VP2, as defined in Figure 6.4, are visualized in the graphic model. In general, a VP constraint is visualized by connecting all join relationships that are associated with the involved decision items. When multiple JRs are associated with one involved decision item, then these JRs are connected with a green dashed line (green color stands for a positive relationship). The graphic VP constraint always is visualized such that the summarized length of lines is the shortest. For example, in Figure 6.5 the left-most of the two JRs associated with D2 was linked

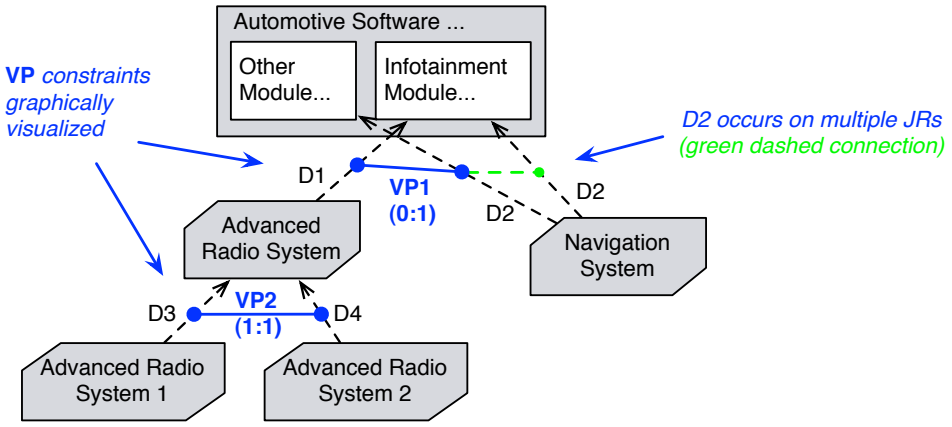


Figure 6.5: A full graphical visualization of the variation point constraints as specified in Figure 6.4 in the graphic SPREBA model of our automotive running example.

for the VP visualization. Further, also the ID and the exact cardinalities are shown together with the VP, to provide comprehensive information. Rationales and other specified information can further be visualized with tooltips, for example.

arbitrary Boolean (C) constraints

In earlier research on variability modeling in software architectures in [Stoiber, 2006], followed by similar experiences in the area of requirements engineering, e.g., [Glinz, 2008b] and [Glinz, 2010a], we have found that other, arbitrary variability constraints (i.e., C constraints) are almost always in the form of implications. Only rarely they are bi-directional implications, respectively equivalences. Therefore, these two operators are used as a basis to define C constraints. The typical form of such a constraint is that some specific variability configuration (an *antecedent*) requires (i.e., *implies*) some other variability configuration (as a *consequent*). The antecedent and consequent terms can be arbitrarily large Boolean formulas, similarly to Batory's definition [Batory, 2005]. They are often only single decision items, though, to define a simple requires dependency between two decision items (Dx $\Rightarrow$ Dy) or a simple excludes dependency (Dx $\Rightarrow$ $\neg$Dy). For every product configuration where the term defined in the antecedent evaluates to $true$, the term specified as the consequent must be satisfied as well. When the antecedent evaluates to $false$, the consequent does not need to be satisfied and becomes irrelevant for the satisfiability of the constraint. To guarantee that a newly defined C constraint does not yield an unsatisfiable product line model during its specification, we create any new C constraint in the form $false \Rightarrow true$. Therefore, regardless of whether the consequent or antecedent is edited first, the constraint is initially always satisfied.

C example

Figure 6.6 shows an example C constraint from our automotive running example. As required in Audi's Q5 online car configurator, the *Bluetooth Hands-free for Mobile Phone* feature (i.e., D5) can only be selected when an *Advanced Radio System* (i.e., D1) or a *Navigation System* (i.e.,

Constraints Table

ID	Design Rationale	Antecedent	Operator	Consequent
C1	Bluetooth Hands-free for Mobile Phone always requires either an Advanced Radio System or a Navigation System to be installed.	D5	=>	(D1 or D2)

Figure 6.6: A constraints table view that specifies a variability dependencies of our automotive running example in Boolean algebra.

D2) is selected. This constraint can straightforwardly be defined as a C constraint with D5 in the antecedent (which triggers the constraint) and D1 *or* D2 in the consequent (which must evaluate to $true$ once the antecedent becomes $true$). This constraint is not contradictive to VP1, which defined that D1 and D2 are mutually exclusive (i.e., with 1:1 cardinalities). Rather, C1 poses an additional restriction to VP1. Figure 6.6 shows how such a constraint is straightforwardly defined in the constraints table, along with a dedicated rationale.

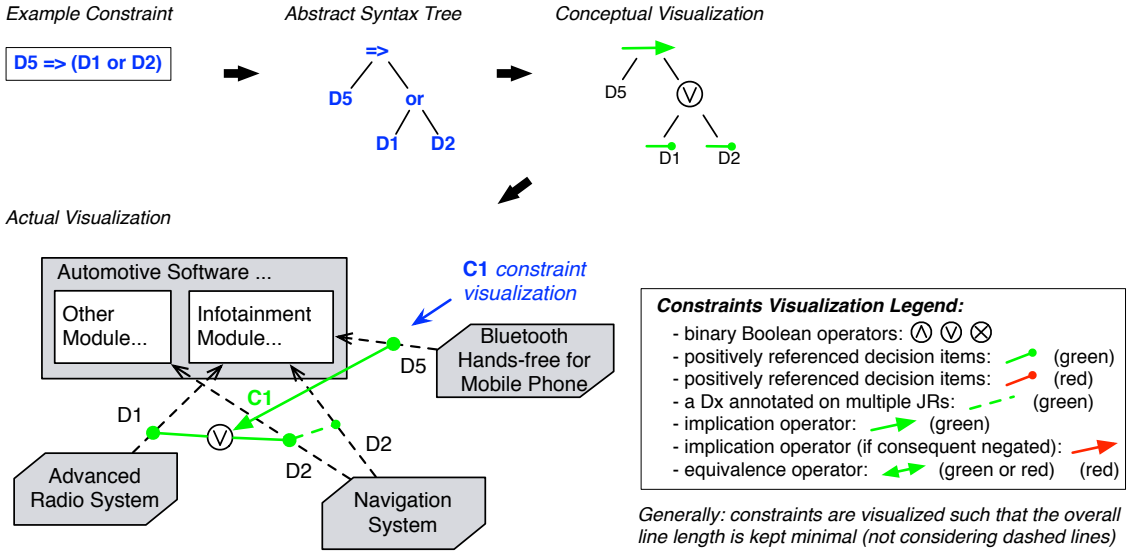


Figure 6.7: An example of how the abstract syntax tree of C constraints looks like and how such a constraint is visualized in the requirements model.

Figure 6.7 shows how the C constraint defined in Figure 6.6 is visualized in the SPREBA model. Note that this SPREBA model has a different topology than the one shown in Figure 5.2, where the constraint C1 was implicitly already specified. Figures 6.7 and 6.6 define the same dependency with a C constraint, though. The root of every C constraint visualization is the logical operator, which is either an implication (=>) or an equivalence (<=>). This operator connects

C visualization

the visualization of the antecedent and the consequent terms. When the antecedent or consequent are only a single decision item, then the visualization is straightforward. Otherwise, the specified terms are visualized by their abstract syntax tree, by visualizing all the binary operators graphically and by directly connecting the JRs of the involved decision items to these operators. Whenever a decision item is annotated to multiple join relationships these are also connected, by a dashed green line, to indicate the multiple associations of this decision item. In general, *green* color is used to visualize positive relationships and *red* is used where a negation is involved. Since a negation of a decision item or of a Boolean term of decision items is always a unary operator, it can directly be mapped to its visual representation, by displaying the connection from or to a negated term in red color. Figure 6.6's bottom-right section provides a comprehensive legend of these visual model elements and their meaning. As with VP constraints, a C constraint is always visualized on those JRs that overall yield the smallest visualization (i.e., in terms of line lengths, not counting the dashed green lines), as a straightforward rule.

selective constraints visualization

The visualization of variability constraints is to date just straightforwardly an additional layer on top of the graphical SPREBA model. Lines that connect the involved join relationships are not routed as intelligently as state transitions or scenario connections are in ADORA [Reinhard, 2010]. The reason for this is that we rather strive for a selective visualization of constraints only when needed, instead of a permanent visualization that is neatly integrated into the graphical model [Stoiber et al., 2008]. Since engineers typically model or comprehend only single decision items or features at a time, only these constraints may be relevant then, which suggests such a selective visualization [Stoiber et al., 2008]. In general, the capability of precisely visualizing any variability constraint exactly as formally defined allows a complete visualization of the variability model and its dependencies. This provides the basis for an efficient and holistic reasoning about variability in the graphic SPREBA model.

6.1.3 Decision Table: Automated Analysis and Configuration

The *decision table* view is a means for three major purposes: (i) it supports the documentation of details about the variability (recall Figure 6.2), (ii) it presents detailed automated analysis results to foster a better understanding of the actual impact of constraints on variability binding decisions, and (iii) it supports setting variability binding decisions to configure application products (although the selection or deselection of such binding decisions could also be realized in the graphic model). Figure 6.8 presents an excerpt of our running automotive example's decision table, which particularly highlights the latter two purposes. It presents details about the *Advanced Radio System* (i.e., D1), the *Bluetooth Hands-free for Mobile Phone* (i.e., D5), and the *Mobile Phone Docking Station with Voice Control* (i.e., D6) features. Figure 6.9, as follows later, shows the complete SPREBA model, where all decision items and their associations are visible.

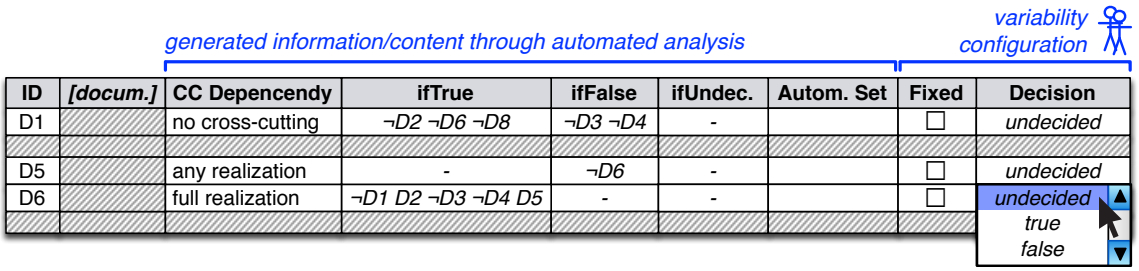


Figure 6.8: A decision table view example that shows all basic columns of the approach and that is easily extensible with further columns.

variability configuration

During domain engineering and when starting a new product derivation all variability is unbound (i.e., neither selected and nor deselected) and, hence, set $undecided$ in the decision table's *Decision* column. When a product derivation was initiated, engineers go forward to configure the variability of a SPREBA model by deciding a single decision item to either $true$ or $false$ as a first step. SPREBA's approach to product derivation, hence, is a stepwise and incremental one, where only a single variability binding decision is taken by an engineer in every single step. A detailed introduction into SPREBA's product derivation capabilities will follow in Chapter 8. Figure 6.8 visualizes how a variability binding decision is taken in the *Decision* column of decision item D6. An engineer has selected the *Decision* field and is about to select or deselect the variability associated with this decision item (i.e., in this case the *Mobile Phone Docking Station with Voice Control* feature). Taking a variability binding decision automatically triggers a constraint propagation, as visualized in the columns *ifTrue* and *ifFalse*, and automatically composes or removes the respective functional specification from the graphic SPREBA model, as further described in Chapter 7. In SPREBA, any variability binding decision can be taken or changed at any time during a product derivation. To avoid overloading the visual representation in the graphic model, the actual decision-taking is realized in the *decision table* view, which is also well suited for the automated analysis results required for a well-reflected decision taking.

fixing decisions

Variability binding decisions in SPREBA can freely be set ($undecided \rightarrow true|false$) and changed when already set ($true|false \rightarrow undecided|true|false$) at any time. When such a change introduces a conflict with other already taken decisions, SPREBA's automated analysis solution automatically finds a minimal set of other decision value changes such that all constraints are again satisfied. How exactly this analysis is performed will be covered in Section 9.4.2 in detail. Engineers typically are very certain about particular variability binding decisions, but rather aim to find an ideal trade-off about others, where they are not so sure yet (i.e., depending on the constraints). Thus, whenever a conflict is resolved automatically, an engineer may not want the decisions he or she already was absolutely certain about to be changed anymore. To improve SPREBA's dynamic variability configuration capabilities in this regard the

Fixed column, where an engineer can lock-in or fixate a decision item to a particular value, is provided. A decision item that has the checkbox in the *Fixed* column marked will not be considered for any minimal change set that is evaluated. Hence, it will always stay on this fixed value, unless the engineers decide to manually change this chosen and fixed decision value. Figure 6.8 shows this *Fixed* column, where a checkbox can be ticked to fix the decision value of any specific decision item.

automated analysis results

The other five columns to the left of the two configuration columns in the decision table (Figure 6.8) are pure projections of the results of our automated analysis. This automated analysis is based on the SPREBA model, all its variability constraints and the currently set variability configuration (i.e., the variability binding decisions and fixations). In the following we will briefly discuss the information that is visualized in these columns. How exactly this information is calculated will be explained in Chapter 9.

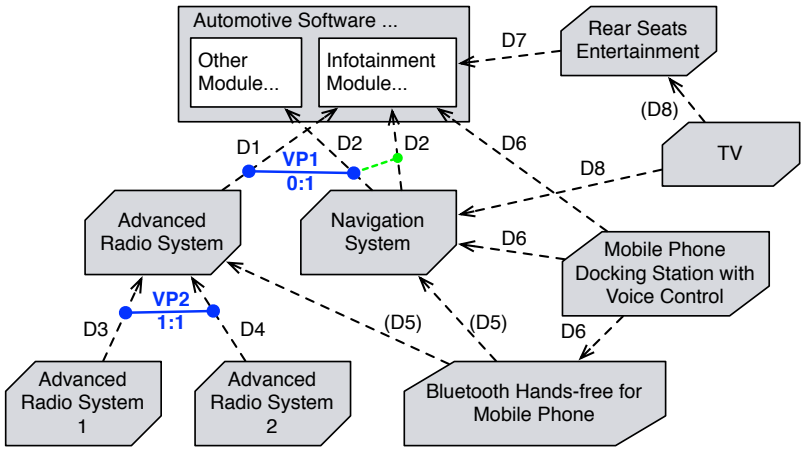


Figure 6.9: A complete but abstract view on our automotive running example that visualizes all variable features, decision items, and constraints.

cross-cutting dependencies

The column *CC Dependency* stands for cross-cutting dependency and presents the extent of realization that is required for a decision item. If a decision item impacts only a single variable feature or the commonality, then *no cross-cutting* will be shown in this column. If a decision item impacts the specification of multiple other decision items, then it becomes relevant whether the decision item is strongly associated with all its JRs or also weakly associated with some or all of its JRs. When it is strongly associated with all JRs, then *full realization* will be displayed. When it is weakly associated to some JRs, but not to all, then *partial realization* is shown. When it is weakly associated to all JRs, then *any realization* is shown. This distinction is important, as it shows to an engineer that a particular decision item's associated requirements may not be completely applied in their full extent, when the decision item's cross-cutting also

allows a partial realization through weak decision item associations. A more detailed and formal specification of these CC dependencies follows in Section 9.3.
Figure 6.8 shows how these CC dependencies are visualized in the decision table for three particular decision items. As shown in the example presented in Figure 6.9, the *Advanced Radio System* feature (D1) is clearly not cross-cutting, since it only impacts the commonality and no other feature. The *Bluetooth Hands-free for Mobile Phone* feature (D5), further, is cross-cutting because it can be applied to an *Advanced Radio System* or a *Navigation System*, but only at least one of the two is required such that the feature is validly applied—see the weak association of decision item D5 to all its JRs in Figure 6.9. This actually requires *any realization* to be shown for this decision item in the *CC Dependency* column in the decision table. Hence, this means that no other impacted decision item is strongly required, but that at least one of them must eventually be selected, for a valid selection of D5. Further, the *Mobile Phone Docking Station with Voice Control* feature (D6) impacts the *Navigation System* and the *Bluetooth Hands-free for Mobile Phone* features and is also cross-cutting. For this feature the decision item D6 is strongly associated to all its JRs. Therefore, it logically requires all its impacted decision items to also be selected, when selected. For this reason *full realization* must be shown in the decision table, see Figure 6.8. Finally, the *TV* feature as already shown in Figure 6.3 (i.e., decision item D8) is another example for a *partial realization*, which is a mixture of the previous two, where only the strongly associated JRs are strongly required and the remaining ones are purely optional, see Figure 6.9.

constraint propagation previews

Similarly to Batory's seminal work, which was focused on feature models, the SPREBA approach is also realized as a so-called logic truth maintenance system (LTMS) [Batory, 2005]. A SPREBA model and any partial or full configuration of a SPREBA model always maintains the satisfiability of every yet unbound variability binding decision by performing a Boolean constraint propagation. Compared to Batory's work, these constraint propagations are already calculated beforehand, for every possible change of every variability binding decision. These results are further displayed in the columns *ifTrue*, *ifFalse*, and *ifUndecided* in the decision table view. Since a decision item is always in one of these states *true*, *false*, or *undecided*, one of these columns is always empty or disabled (i.e., because the required propagation is already set). How exactly these propagations are calculated for a SPREBA model is presented in Chapter 9.
Figure 6.8 shows an excerpt of a configuration where all decision items are yet set *undecided*. It shows the required constraint propagations for the decision items D1, D5, and D6, as they are required by the hierarchies, the cross-cutting, and the additional constraints. Figure 6.9 shows the full SPREBA model. For a selection of the *Advanced Radio System* (i.e., when setting decision item D1 to $true$), for example, a deselection of the decision items D2, D6, and D8 is required. This is because the advanced radio system is an alternative to the navigation system, which must be deselected, hence. Further, the mobile phone docking station feature (D6) and the TV feature (D8) can only be realized when a navigation system is chosen and, thus, must also be deselected. For a deselection of the *Advanced Radio System* (i.e., D1 $= false$), a de-

selection of its two sub-features *Advanced Radio System 1* and *2* (i.e., D3 and D4 $= false$) is required. Similarly, when a feature already was selected or deselected and an engineer later considers reverting such a decision back to *undecided* (which formally means that both selecting it and deselecting it must be satisfiable), this may also require a constraint propagation, which is shown in the *ifUndecided* column.
Note that the presented constraint propagations are verified and exhaustive. Section 9.4.2 presents the details of how they are calculated. This means that every undecided decision item that is not listed in a constraint propagation can still freely be taken afterwards. Those decision items listed as part of the constraint propagation must be set this way. Otherwise, a constraint violation is certain and no valid product (i.e., no product configuration that satisfies all the specified variability constraints) can be derived anymore, without changing any of the already bound decision items (i.e., those set $true$ or $false$).

propagated decisions

Finally, it may be important for engineers to keep track of which decisions they took manually and which were set automatically by the tool, as a constraint propagation. This information is displayed in the *Automatically Set* column in the decision table view. This column is essentially a checkbox that is automatically filled-in by the tool. Whenever a decision was propagated, this field contains a check mark for this decision item. Consider for example selecting the decision item D1 to $true$ in Figure 6.8. This requires taking three other variability binding decisions as listed in the *ifTrue* column of the table. After setting D1 $= true$, thus, four decisions will be bound. The decision item D1 will be set manually and contain no check mark in the *Automatically Set* column. The other three decisions, which were automatically taken by the tool as a constraint propagation, will have an explicit check mark in the *Automatically Set* column. During a product derivation these check marks can become important when application engineers deal with large sets of decision items and are reasoning about changes to a given configuration. In such a case, when considering a change, it may be crucial to know whether the concerned decision item was automatically set by the tool, or set manually by a human engineer. Manually changing an automatically set decision would lead to a conflict in the configuration. A manually set decision, on the other hand, is still likely not to cause a conflict and will possibly also undo or change any previous propagation that the original decision triggered. A detailed explanation of the required SAT-based automated variability analysis, to derive this information, will follow in Chapter 9.

6.2 Extended Example and Discussion

Our running example—Audi's Q5 infotainment system, as introduced in Section 5.2—has provided sufficient real-world software product line data to illustrate all the hitherto SPREBA variability modeling concepts. From the real-world data of Audi's online car configurator we identified several cross-cutting features that impact several other features (e.g., the *Mobile Phone*

Docking Station with Voice Control or the *Bluetooth Hands-free for Mobile Phone* features). We also identified variation point constraints and could demonstrate an arbitrary Boolean constraint. To further demonstrate how a variable feature can be configured more precisely with multiple decision items we extend this real-world example with a hypothetical addition.

Extended Example

extending the example

To provide an example for the remaining important special cases that SPREBA is capable of modeling, we introduce the additional, imaginary feature *Data Logging*, which is constructed and *not* a real-world feature in Audi's car configurator, as all hitherto features were. We assume that adding a 'data logging' feature would require a logger component in the infotainment module to write log entries. Further, we assume that the behavior of several other components like the *TV*, the *Mobile Phone Docking Station with Voice Control*, and the *Bluetooth Hands-free for Mobile Phone* features need an extended behavioral specification to write log entries, when particular events occur. The logger component should only be realized when the behavior to write log entries is actually part of the product. On the other hand, a selection of data logging should not have a strong dependency to any of the features where logging can be realized. But when all of the features that could realize data logging are deselected, then also data logging must be deselected and the logger component must not become part of the product.

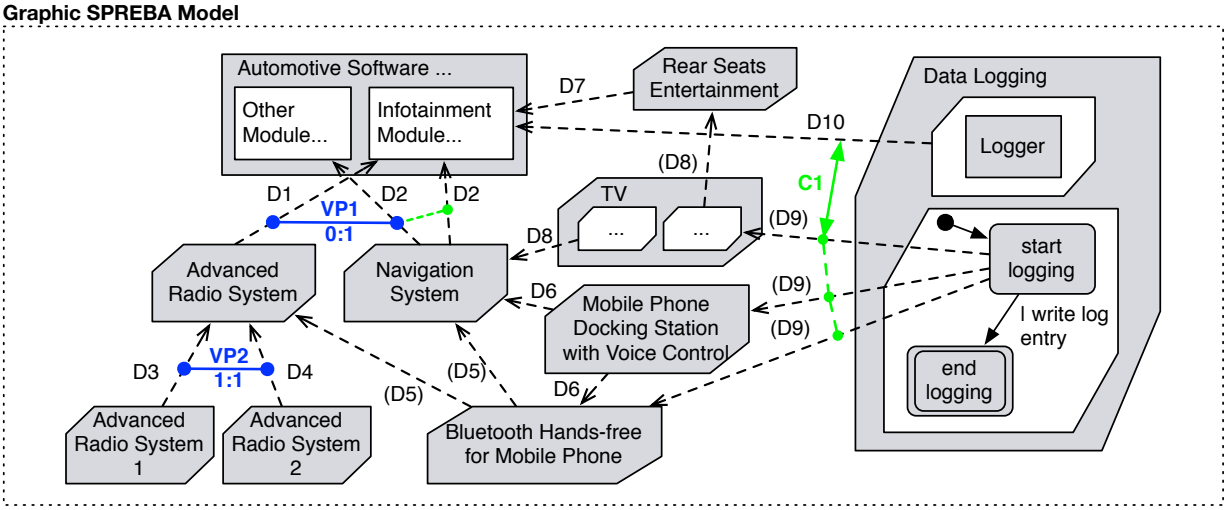
multiple decision items per feature

In Section 6.1 all variable features had a one-to-one mapping to a dedicated decision item. Thus, the decision items could de-facto be used like features in a feature diagram. This is the typical case and also the ideal case, since it fosters simplicity and keeps maintenance activities and configuration simple. However, sometimes the variability in a product line is more fine-grained than a mere selection or deselection of features, with a single dedicated decision item for each feature. To be able to fully cover all possible cases, the use of multiple dedicated decision items for the more fine-grained variability specification of one variable feature is also allowed in SPREBA. An example for such a feature is the data logging feature.

an example

Figure 6.10 shows a complete but abstract view on the specification of our extended automotive running example. The shown SPREBA model focuses on the variability and hides the detailed requirements specifications of the variable features, except for the *Data Logging* feature. It also shows a complete view of the decision table (except for the various documentation columns, for brevity) and the variation points and constraints tables. The *Data Logging* feature shows its full specification, as far as it is modeled. It contains two parts of a feature: the behavior chunk that initiates the writing of log entries after particular events occur in other features and the *Logger* component, which is the functionality that actually writes the log entry. In order to precisely model the required variability constraints this feature actually requires two dedicated decision items. First, the logging behavior chunk (the part of the feature associated with decision item D9) needs to only have weak dependencies to all its target features. This

Graphic SPREBA Model



Decision Table

ID	[documentation...]	CC Depencency	ifTrue	ifFalse	ifUndec.	AutoSet	Fixed	Decision
D1		*no cross-cutting*	*¬D2 ¬D6 ¬D8*	*¬D3 ¬D4*	-		☐	*undecided*
D2		*no cross-cutting*	*¬D1 ¬D3 ¬D4*	*¬D6 ¬D8*	-		☐	*undecided*
D3		*no cross-cutting*	*D1 ¬D2 ¬D4 ¬D6 ¬D8*	-	-		☐	*undecided*
D4		*no cross-cutting*	*D1 ¬D2 ¬D3 ¬D6 ¬D8*	-	-		☐	*undecided*
D5		*any realization*	-	*¬D6*	-		☐	*undecided*
D6		*full realization*	*¬D1 D2 ¬D3 ¬D4 D5*	-	-		☐	*undecided*
D7		*no cross-cutting*	-	-	-		☐	*undecided*
D8		*partial realization*	*¬D1 D2 ¬D3 ¬D4*	-	-		☐	*undecided*
D9		*any realization*	*D10*	*¬D10*	-		☐	*undecided*
D10		*no cross-cutting*	*D9*	*¬D9*	-		☐	*undecided*

Variation Points Table

ID	Design Rationale	minCard	maxCard	DecisionsInvolved
VP1	The Basic Radio System is Commonality and can be extended to either an Advanced Radio System or a Navigation System (which includes all functions).	0	1	D1, D2
VP2	There are two variants: an intermediate (#1) and a high-end system (#2).	1	1	D3, D4

Constraints Table

ID	Design Rationale	Antec.	Op.	Consequ.
C1	The Data Logger (D10) component needs to be realized when data logging (D9) is realized anywhere in the system, but must not be realized otherwise. Therefore, two separate but equivalent decision items are needed.	D9	<=>	D10

Figure 6.10: A complete view on our automotive running example that also includes an imaginary additional feature for *Data Logging*. The Figure shows the graphic SPREBA model, the decision table view, and both constraints tables.

formally requires that at least one of these target features needs to become part of the final product, such that the logging behavior chunk also does. Second, the logger component needs a separate decision item because it applies directly to the commonality and would, thus, void the *any realization* cross-cutting dependency if it used the same decision item. This is because *any realization* would always be satisfied when one JR directly impacts the commonality, since

the commonality is always $true$ and part of the product. As shown in Figure 6.10, D9 has been introduced for the logging behavior chunk and D10 was introduced for the logger component. D9 is weakly associated with all its JRs, while D10 is strongly associated with its JR. As the JR of D10 directly impacts the commonality, the type of decision item association actually does not matter in this case and becomes a strong association by default (i.e., both would have the same semantics in this case). To assure that the logger component never is applied without any logging behavior applied somewhere in the system, a Boolean logic constraint has been added, see constraint C1 in Figure 6.10. This constraint requires that whenever the logging behavior is chosen the *Logger* component will also be added to the product. Vice versa, whenever the *Logger* component is chosen also the logging behavior needs to become part of the application product somewhere—otherwise the *Logger* component will not be used and must be deselected. On a side note, an alternative design with dedicated decision items for adding logging to every of these targeted components could also be defined. For example, (D9) could be split into the three separate decision items D9, D11, and D12, which could replace the three annotations of (D9) in Figure 6.10. The constraint C1 could then be refined into D10 $<=>$ (D9 $\vee$ D11 $\vee$ D12). This would yield a slightly different variability specification, which would also only allow to choose logging when at least one of these parent features is chosen. Additionally, it would allow to separately select and deselect logging for every possible parent feature. SPREBA allows such a precise and fine-grained specification of both of these design options. The subtle differences in terms of variability dependencies will automatically be analyzed exactly as specified.

The *Data Logging* feature is special because it is one feature but involves more than one decision item. In the graphic SPREBA model it was encapsulated by one comprehensive feature container, called *Data Logging*. It contains two parts of a feature that need to be handled with separate decision items, however. It also contains a feature-internal variability constraint C1. While it overall is one feature, it requires two separate variability decisions that are highly constrained, in order to precisely and adequately model the required variability. While in the abstract view presented in Figure 6.10 such a more complex situation only occurred in the *Data Logging* feature, such additional complexity is also likely to occur in other features, once the specification is evolved to a significantly more detailed level. For example, the *Mobile Phone Docking Station with Voice Control* is possibly much more complex when specified at a very fine-grained level (compared to the model shown in Figure 5.2).

Discussion

non-trivial constraints occur

The features selected from Audi's online car configurator for the Q5 infotainment system contain a significant portion of constraints. When these constraints are specified in an accurate and formal form, then they quickly become non-trivial to comprehend and to verify manually, especially when they transitively interact with each other. As shown by the columns *ifTrue* and *ifFalse* in Figure 6.10's decision table, the variability model is already non-trivial for this

small example (only eight decision items, or ten, when also considering the data logging). In fact, our automotive running example includes only one feature that is purely optional, which is the *Rear Seats Entertainment* (D7). Although, even this feature can be realized either with or without TV support. However, it does not require or imply any other variability configuration, neither directly nor indirectly, because of the weak dependency and the CC dependency *partial realization* of D8. Especially transitive relationships between multiple variability constraints quickly become hard and unintuitive to comprehend for human engineers. Therefore, tool support for an automated analysis and an automated calculation of constraint propagation previews (e.g., as shown in the *ifTrue* and *ifFalse* columns and more) can have a profound potential for improvements and understanding during product configuration activities.

differences to feature modeling

Compared to existing variability modeling languages (recall Sections 3.2 and 3.3), and feature modeling in particular (Sections 3.2.1 and 3.3.1), the SPREBA notation is significantly more expressive in the following three regards—from a language point of view. These constitute the main differences to feature modeling, in our opinion (as also reflected in Section 1.2's technical contributions):

1. *Explicit specification of cross-cutting features.* The variability in a SPREBA model can also specify any arbitrary cross-cutting between variable features. This specification of cross-cutting is a direct result of functional dependencies between features (recall the *TV* feature, for example). A feature model, in contrast, is always defined in a pure tree format. In a feature model such dependencies, thus, would need to be specified with additional cross-tree constraints, to assure correctness of the product line model and its configurations. There are ideas about modeling aspectual features, e.g., [Czarnecki et al., 2005b, Figure 1]. However, to the best of our knowledge, there is no systematic approach that also allows weak dependencies or similar, which is required to adequately and precisely define the variability of particular cross-cutting features, like for the *TV* or the *Bluetooth Hands-free for Mobile Phone* features, for example. Modeling such cross-cutting features directly as aspects may yield feature hierarchies and dependencies that might be closer to the actual dependencies that exist in a software product line. Our automated analysis solution provides full support for such cross-cutting features and weak feature dependencies, see Chapter 9.

2. *Systematic support for arbitrary variability constraints.* Feature models and OVM models often allow only simple cross-tree variability constraints like requires and excludes dependencies. Batory, for example, has claimed that these are too simplistic [Batory, 2005]. While most research-focused tools still use only these simple constraints, many industrial-strength tools today already allow more complex ones, however. In general, variability constraints also depend on the features they are defined upon. For example, when D1 is set $false$ in Figure 6.10, then the constraint VP2 is rendered irrelevant because all its involved decision items do not have a so-called *commonality path* ($cpath$ in short, as follows in Chapter 9) anymore, and both D3 and D4 will also be set $false$. A

variation point constraint becomes irrelevant when all its involved decision items do not have a satisfiable $cpath$ predicate. An arbitrary Boolean constraint becomes irrelevant when either all its decision items used in its antecedent (i.e., in case of an implication) or all its decision items (i.e., in case of an equivalence) have an unsatisfiable $cpath$ predicate. Since SPREBA provides a complete and systematic solution for handling hierarchies and cross-cutting (i.e., through the $cpath$ concept, see Section 9.3), it also provides a full solution for handling the relevancy of constraints. Therefore, it supports the specification and automated analysis of arbitrarily complex Boolean variability constraints as well. A detailed description of SPREBA's automated variability constraints analysis is presented in Chapter 9.

3. *Integration of variability and requirements model.* A SPREBA variability model does not require any mappings to other requirements artifacts, but *contains* these requirements artifacts. This is—to the best of our knowledge—in contrast all existing variability modeling approaches, recall Section 3.2.2. Functional dependencies between requirements artifacts of different features, thus, will directly be modeled as cross-cutting features. This can avoid costly errors by missing particular variability dependencies already beforehand, during the variability model's construction. While existing approaches require traceability matrices, presence conditions, or feature annotations in other requirements descriptions, SPREBA provides an integrated notation, which uses view generation. This allows generating abstract, pure variability model views on any software product line specification, but also allows an integrated visualization of the actual functional requirements, whenever desired. Hence, compared to feature modeling and the use of a mapping-based approach, SPREBA fully integrates the requirements and variability model into a single integrated and coherent visual notation.

CHAPTER 7

Feature Weaving

A SPREBA product line domain model represents all variable features as aspects (i.e., in a compositional form) and only the commonality as a plain model. In a derived application product model, where all variability binding decisions are taken, no variable feature will be represented by aspect-oriented modeling anymore. All selected features will be woven (composed) and all deselected features will be removed. During a product derivation, however, only some decision items' join relationships need to be composed (i.e., those already selected), some removed (i.e., those already deselected), and some still visualized as aspect-oriented modeling (i.e., those yet left undecided). Further, any arbitrary changes or reversions of decisions in a given configuration must be processed by this solution as well. This goes beyond classic aspect weaving. A refined solution is necessary, to (i) provide views on any partially derived SPREBA models that are consistent with any currently set (partial) variability configuration and to (ii) maintain the hitherto mental map of the model at the same time. In the context of SPREBA such a solution has been developed and is called *feature weaving*, as introduced in this chapter.

7.1 From "Static Weaving" to Dynamic Weaving

limitations of static weavers

A classic or static aspect weaver essentially composes an aspect-oriented model (i.e., a model that contains aspect-oriented modeling) into a woven model (i.e., one where all aspects are composed). A static aspect weaver, however, is *not* capable of composing only a subset of aspects

at a time, of undoing only a subset of the previous weaving operations, and of re-visualizing the aspect-oriented modeling of particular aspects that were previously woven. These operations must be supported to realize SPREBA's stepwise, incremental product derivation, however, as follows in Chapter 8. A static weaver for ADORA has been presented in [Meier, 2009]. For most product line engineering-related tasks relying only on such a static aspect weaver is insufficient, though. A significantly refined version of the weaver is required, which is capable of realizing *feature weaving*. Our efforts to realize such a feature weaver initially begun by developing more flexible and selective weaving mechanisms, which we originally called *dynamic weaving* [Kandrical, 2009]. Such a flexible approach to aspect weaving is required in both domain engineering and product derivation. To systematically support *product derivation* (Chapter 8) the weaver needs to be capable of composing and/or removing particular variable join relationships as soon as a new variability binding decision is taken, without changing the rest of the model. During *domain engineering* the desired feature weaving solution needs to be able to turn single extracted variable features into mandatory features by selectively weaving these model elements and removing the aspect-oriented modeling. This is particularly necessary to support the evolution of SPREBA domain models. Additionally, the weaving must be performed in the currently displayed view and configuration, to make sure that the mental map of the model is preserved between all subsequent variability configurations. Not preserving the mental map of the model whenever the variability configuration changes would hinder the understandability considerably, as the model's overall layout would differ every time the variability configuration changes, even for those parts that did not change. The reason is that a selective aspect weaver (i.e., a static weaver that weaves only some selected aspects and not the others) would need to re-weave all aspects again for every new or changed configuration.

Meier's static weaver

Meier has already developed a static weaver for ADORA in previous work [Meier, 2009] (Section 2.4.2). On the one hand, this weaver satisfies all the basic prerequisites for using the SPREBA approach, as described in Chapter 5.1. On the other hand, however, using Meier's static weaver can not support the tasks highlighted in the previous paragraph. Meier's aspect weaver merely supports switching between fully aspect-oriented and fully woven views. A particular difficulty of Meier's solution further was the creation of a model copy (i.e., a duplicate of the ADORA model) to perform the aspect weaving and to present the woven view. Weaving aspects only in a model copy did not allow any undo operations of variability binding decisions, except for restarting the derivation process at the original domain model and re-composing all selections again. This typically resulted in a quite different graphic layout every time a variability binding decision was changed and, hence, the mental map of the model was lost with every change in the variability configuration. Our work on SPREBA initially relied on Meier's weaving solution [Kandrical, 2009]. This was highly undesirable, though, because not all arbitrary changes in the variability configuration could easily be generated and because the mental map of the model was disturbed too often. Therefore, an aspect weaving solution was needed that systematically handles all the required model transformations and preserves the mental map over the whole process at the same time, as follows.

dynamic weaving

A dynamic weaver must be capable of weaving, undoing a previous weaving, removing/hiding, and re-visualizing parts of the aspect-oriented model (i.e., join relationships and the associated aspects, respectively), when required. The term dynamic weaving, in the context of ADORA, was first introduced in [Kandrical, 2009]. Every time a variability binding decision is taken (i.e., an $undecided$ decision item is set $true$ or $false$) all associated join relationships and aspect(s) must instantly be composed (when set $true$) or removed from the model (when set $false$) and all other aspects must remain unchanged. Importantly, these dynamic weaving operations must also happen in the same model because any subsequent weaving operations must be realizable based on this just generated new layout and because undoing operations must also be supported. When an engineer undoes any already set variability binding decision (e.g., from $true$ back to $undecided$), then the previously woven model elements need to be removed again and the respective original aspect-oriented modeling needs to be re-visualized. We initially called an aspect weaver that realizes all these requirements a *dynamic weaver*. The term "dynamic weaving" has been used before, however. Popovici et al., for example, introduced dynamic weaving for selectively weaving aspects into running Java programs [Popovici et al., 2002]. Further, Hallsteinsen et al. introduced the concept of dynamic software product lines, which targets at realizing changes in the variability not only before the product's deployment, but also during the runtime of an application product [Hallsteinsen et al., 2008]. These approaches also allow the composition of only single aspects at a time. But they plainly focus on systems at runtime. In the here presented approach dynamic weaving is particularly required to generate consistent views on partially derived conventional product line requirements models (i.e., with an integrated concrete syntax and visual notation). A dynamic weaver needs to support domain model refactoring and the product derivation process for a SPREBA product line model.

feature weaving

We use the term 'dynamic weaving' only within this chapter to illustrate the general concept of *feature weaving*. In particular, when variability constraints are involved we speak of feature weaving. This is because also the visualization of variability constraints needs to be adapted when decision items involved in variability constraints are bound (as follows in Section 8.2).

7.2 Realizing Feature Weaving

realizing feature weaving

To realize feature weaving, the weaving of variability needs to be done in the *same* model, in order to enable the creation of all potentially reachable views and to maintain the mental map as far as possible. Feature weaving must allow a selective weaving of single join relationships, such that the graphically visualized requirements model is always consistent with the current configuration of variability binding decisions. These decision item bindings, for which the SPREBA model needs to be adapted in the graphic view on the requirements model, are set in the decision table view. Whenever any variability binding decision is changed (i.e., any decision item d with $d.val \in \{true, false, undecided\}$ is changed to $d.val' \in \{true, false, undecided\}$

where $d.val' \neq d.val$), the feature weaving function is responsible for creating a view on the graphic requirements model that presents all associated join relationships either woven (for the decision items set $true$), removed from the model (for the decision items set $false$), or visualized as aspect-oriented modeling (for the decision items set $undecided$). Therefore, all associated join relationships and aspects for every decision item need to be handled as follows, depending on a decision item's truth value:

1. *undecided:* all associated JRs and ACs need to be displayed as plain aspect-oriented modeling; if these model elements were woven before, all these woven elements need to be removed,
2. *true:* all associated JRs need to be woven; this composes all variable elements into their target locations and the associated JRs and all further associated ACs are not visualized anymore, or
3. *false:* all associated JRs and their respective ACs need to be hidden from the model; if these model elements were woven before, all the woven model elements must be removed.

an illustration

Figure 7.1 shows an example ADORA requirements model with one optional feature (this model was also used in [Stoiber and Glinz, 2010b]) and illustrates how the requirements (i.e., the JRs, the ACs, and the contained variable model elements) need to be presented when the involved decision item D1 is set either $undecided$ (top), $true$ (bottom left), or $false$ (bottom right), as specified above. In this example there is a one-to-one mapping between the feature and decision item D1, which implies that the two are synonymous (i.e., D1 equals to the selection or deselection of *Feature A*).

processing decision item bindings

To implement feature weaving with a tool, an automatic processing of all possible changes of variability binding decisions needs to be supported, as illustrated in Figure 7.1. When the variability binding of a particular decision item Dx changes, four different types of model transformations are necessary to process such a change. These are the following:

- weave all JRs associated with Dx (the source elements of these JRs and the remaining advice needs to be woven—see Section 2.4.2),
- hide all JRs associated with Dx and also all hide respective ACs (every AC that has no unhidden JR left is irrelevant to the model and must also be hidden),
- remove all woven model elements that originated from a previous weaving of JRs that were associated with Dx (every model element that is woven by setting a decision item Dx to $true$ is annotated with an attribute $origin$ set to Dx; removing of woven elements relies on the feature unweaving function, as described in Chapter 10),
- unhide or re-visualize all JRs associated with this Dx and their respective ACs (the aspect-oriented modeling is not really deleted by a weaving operation, but still remains hidden or filtered out of the visualized view, respectively, until a full product configuration is

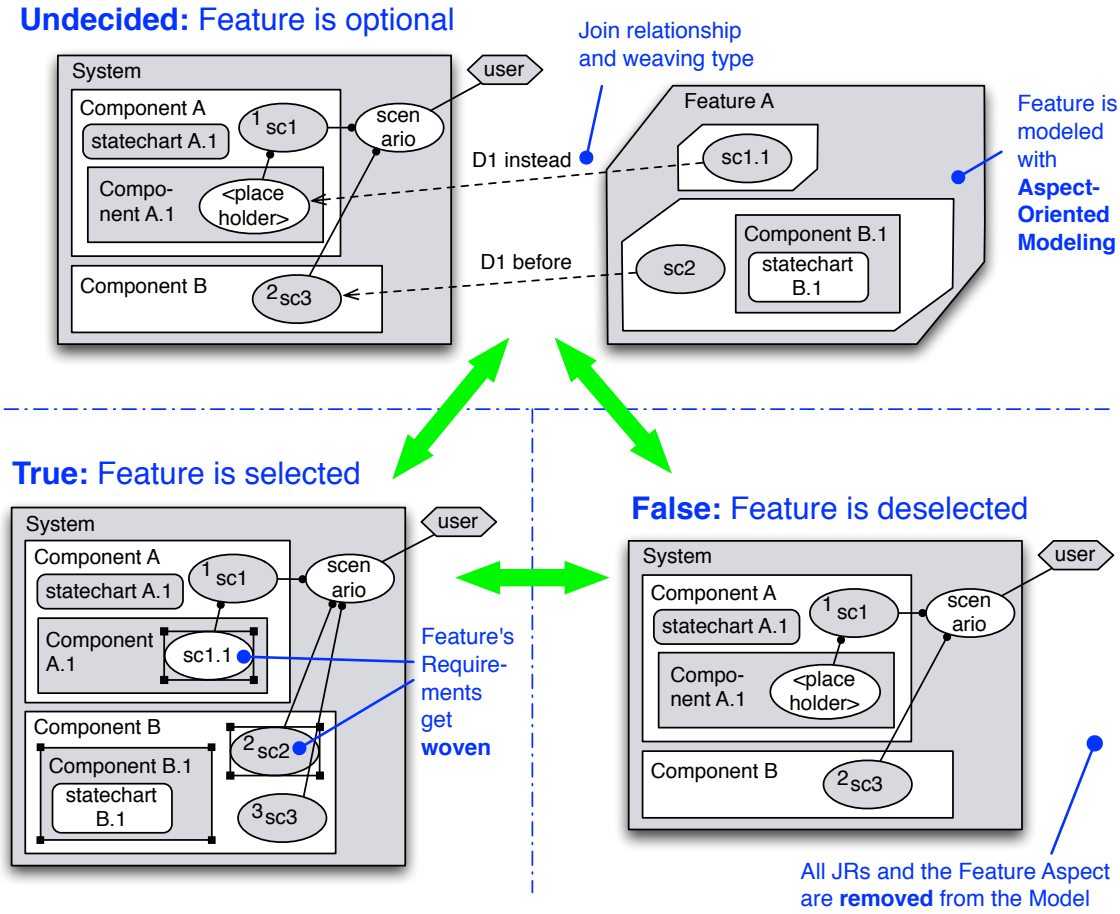


Figure 7.1: An illustration of *feature weaving* for an ADORA and SPREBA model with one variable feature; when the feature is *undecided* it needs to be visualized as a plain aspect (top); when it is selected, it needs to be woven (bottom left); and when it is deselected, it needs to be removed (bottom right). Any change must be possible for any decision item binding.

reached and agreed; only then a permanent removal can be done, along with a final optimization of the model's graphic layout; until this eventual finalization of the product model any arbitrary change and reconfiguration of the hitherto derived product model is still possible).

Figure 7.2 further shows the required behavior of a tool that implements *feature weaving*, based on these essential types of model transformations. As an input this model transformation requires all decision items that changed (i.e., the user's manual variability binding decision plus its constraint propagations), their previous decision value $d.val$, and their new value $d.val'$. For every of these decision items the tool's behavior, as shown in Figure 7.2, will be executed fully

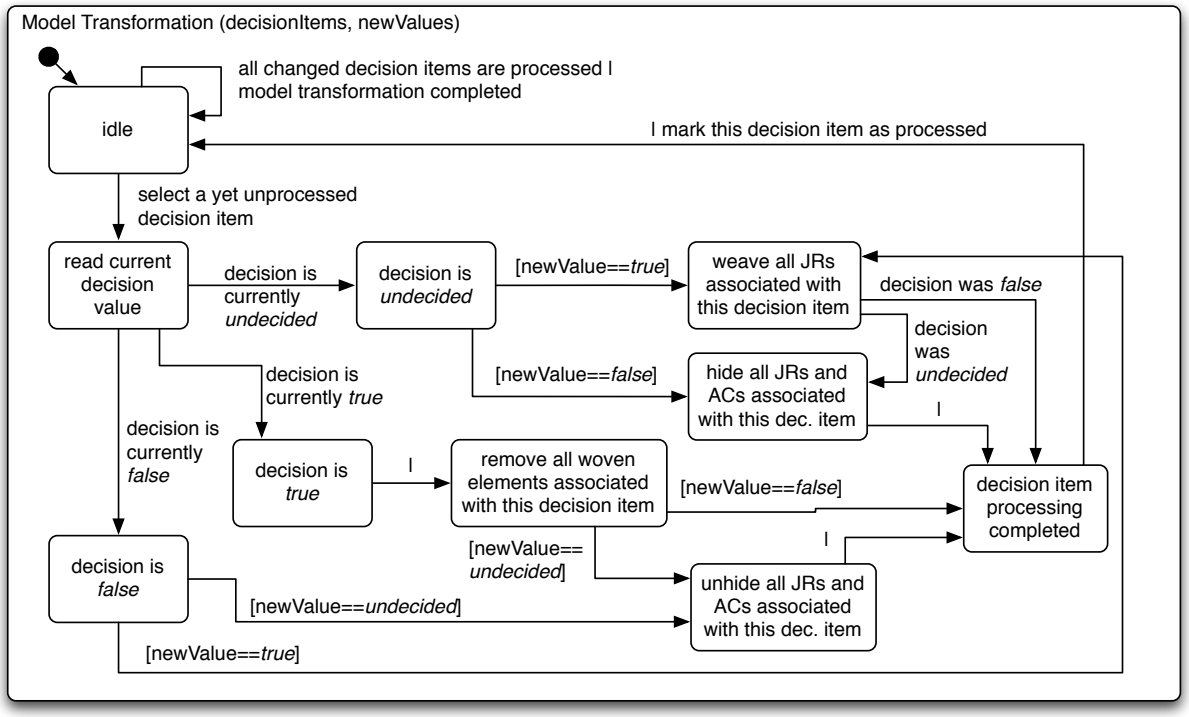


Figure 7.2: A specification of a tool's required behavior to perform a model transformation that realizes a *feature weaving* of changes of one or more decision item's truth values.

automatically. The Figure specifies how the four essential model transformation operations need to be performed, depending on the change of a particular decision item's value ($d.val \rightarrow d.val' \mid d.val' \neq d.val$). The ability to process such arbitrary decision item changes allows the consistent generation of any partially or fully woven view of a SPREBA product line model. Consistency of partially derived product line models, however, also requires the variability constraints to be re-visualized in a derived form, when some of their involved decision items are already bound. This required behavior for constraint derivation and re-visualization is part of SPREBA's product derivation capabilities, however, and follows in Section 8.2.

views on different levels of abstraction

ADORA models can be visualized at different levels of abstraction. Single abstract objects, states, or aspect containers can be collapsed with fisheye zooming (i.e., vertical abstraction) and particular types of model elements can be filtered out of the whole model (i.e., horizontal abstraction), recall Section 2.3.3. ADORA's aspect modeling approach can also visualize the aspect-oriented model and woven models on any such arbitrary level of abstraction, recall Section 2.4.2. The feature weaving approach maintains this capability of visualizing models on a more abstract level. When the contents of a target model element of a specific decision item are not visualized, then the dynamic weaving is performed equally on the underlying model—the

resulting view will merely not highlight the actually composed model elements, but only the component within which they were composed, instead. When the contents of an aspect container are not visible, but the details of the target container are, then the woven model elements will be visible after the composition (Figure 2.11). Thus, in general, a feature weaving operation, as specified in Figure 7.2, will always keep the currently generated view on the model as a whole. Since this view is essentially based on the model's hierarchical topology (recall Figure 2.6), maintaining this topology is rather straightforward. Keeping particular types of model elements filtered from the currently shown view (i.e., horizontal abstractions, recall Figure 2.8) is also straightforward.

positioning woven elements

In the ADORA tool the weaving of aspects is performed fully automatically. The tool also has to position the newly added, woven elements fully automatically. Currently, we still use Meier's solution for positioning newly woven model elements as shown in Figure 2.10 [Meier, 2009]. While this solution is reasonable for weaving behavior or scenario chunks, to some extent, there is no optimal solution for composing what Meier called "embedded components", which are independent model elements that are not integrated into existing state- or scenario charts. This problem of creating optimal and human-friendly layouts when graphically composing aspects has been addressed in [Kandrical, 2009], along with realizing a dynamic weaving solution for ADORA. However, Kandrical could not come up with a satisfying, constructive solution for such cases, either. Therefore, to date, ADORA still relies on [Meier, 2009]'s solution, where unconnected model elements merely are positioned below all the remaining modeling in a specific parent container, when composed. An example of this is shown in diagram *c)* in Figure 2.10. Finding an optimal solution for automatically arranging layouts that are really human-friendly, that keep the mental map of the model as far as possible and that also are overall efficient and compact, without overlaps or similar, is a hard problem. The automated generation of ideal woven layouts, hence, is still left open for future research. Finding an ideal layout means finding an ideal trade-off between many variables. This problem is possibly of an intractable nature in ADORA and SPREBA models.
Further, a model's ideal layout may also differ considerably by the cultural background of the engineers working with it. Reinecke, for example, has shown that the automated re-arrangement of graphical user interfaces can considerably improve the users' performance and satisfaction, when using a culturally adapted version of a deployed application's user interface [Reinecke, 2010]. We believe that similar efforts could also considerably improve working with graphical models like SPREBA models. Such culturally adaptive model layout optimization, however, is also out of scope of this thesis and left open for future research.

CHAPTER 8

Stepwise, Incremental Product Derivation

state-of-the-art solutions

The automatic generation of software products from reusable assets yields the major benefits of software product line engineering. In state-of-the-art approaches actual software products are derived by performing two major activities. First, a *variability configuration* is specified as an instance of the variability model, with all variability bound, respectively selected or deselected. This variability configuration typically ensures that all variability constraints are satisfied, recall Section 3.3. Second, based on this variability configuration, a *product specification* (i.e., the requirements model of a software product) is *generated automatically*. This is done by either pruning all not selected variability, by composing all selected variability, or similar (i.e., depending on the chosen variability realization mechanisms, recall Section 3.2.2).
This separation of variability configuration and product generation leads to conceptual problems in understanding the impact of variability binding decisions, as we motivated in Section 4.3. While advanced tool support that uses colors (e.g., like in [Czarnecki and Antkiewicz, 2005] or [Kästner, 2010]) and on-the-fly product generation during variability configuration could alleviate this problem, these existing approaches still require a user to switch between several separate diagrams for a full impact comprehension.

stepwise, incremental product derivation

In a SPREBA software product line model the variability and the requirements model are specified in an integrated diagram and a compositional approach is used for variability modeling. This allows a seamless integration of the variability configuration and the automatic product generation, as demonstrated in this chapter. This integration relies on two major components: (i) feature weaving and (ii) an automated constraint propagation that ensures that every partially

or fully derived model is always consistent with all variability constraints. Feature weaving has already been introduced in Chapter 7. The automated analysis of SPREBA variability models, which includes the calculation of constraint propagations, is based on SAT solving and is presented in Chapter 9.
SPREBA's stepwise, incremental product derivation has originally been presented in [Stoiber and Glinz, 2010b]. This publication, however, did not yet describe how *conflicts* caused by manual changes of previously propagated variability binding decisions are *resolved automatically*. Such an automated conflict resolution has actually been mentioned as an unsolved problem in recent literature, e.g., [Nöhrer and Egyed, 2010]. This chapter comprehensively illustrates the behavior of a tool that is required to implement a stepwise, incremental product derivation that can generate *consistent* views on the requirements model for *any* change in the variability configuration. Further, it also presents an example product derivation that illustrates the approach. The details of how the constraint propagations are calculated based on a SAT solver will follow in Chapter 9.

8.1 Integrating Configuration and Product Generation

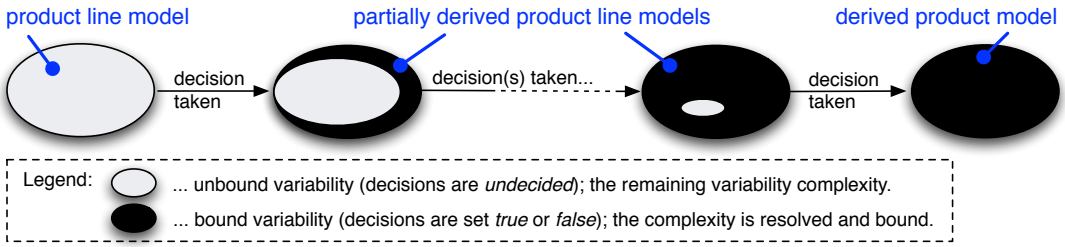


Figure 8.1: A simple illustration of how stepwise, incremental product derivation reduces the variability complexity with every variability binding decision ($undecided \rightarrow true|false$) into a less complex product line model and eventually a product model.

stepwise refinement

The basic idea of stepwise, incremental product derivation is to support a stepwise refinement of the product line model into a product model. Every time a variability binding decision is taken (i.e., a decision item is set from $undecided$ to either $true$ or $false$) also the constraint propagation of this decision is set, all selected join relationships and aspects are composed and all deselected ones are removed. This is realized with feature weaving (Chapter 7). Figure 8.1 illustrates this basic idea of stepwise refinement of the requirements model during a product derivation, where a single variability binding decision is manually taken with every step. Every time a new, unbound variability binding decision is bound (i.e., selected or deselected), the resulting model will be a reduced, less complex remaining product line model or a product

model (i.e., when all variability decisions are already bound). Since the variability constraints are formally analyzed and propagated (as presented in the decision table's columns *ifTrue* and *ifFalse*), all reachable partially derived product line models are always fully valid and consistent.

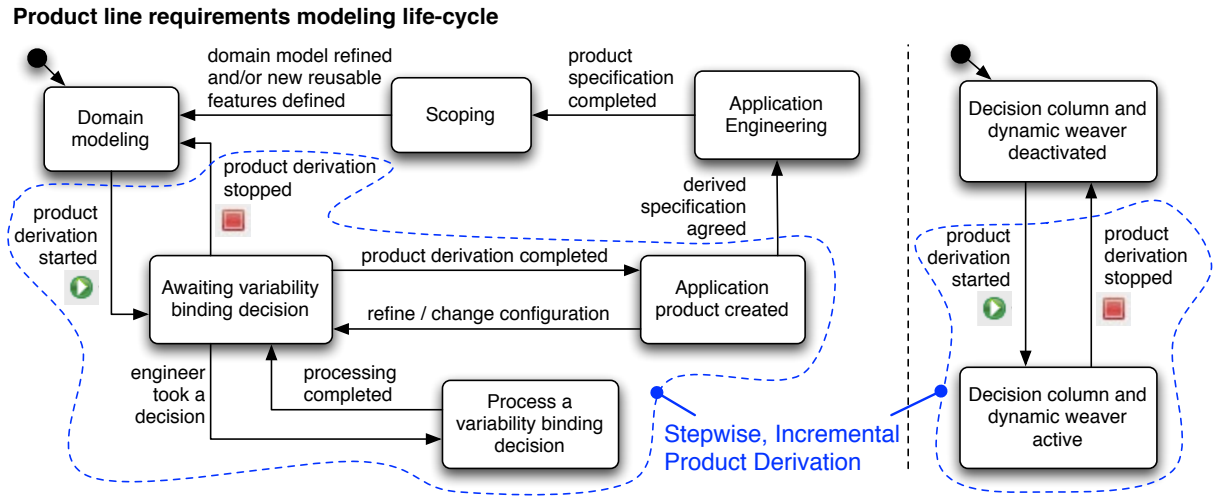


Figure 8.2: An overview of how stepwise, incremental product derivation integrates into the overall software product line life-cycle.

product line life-cycle

The initial activity in a software product line engineering process in general is the development of a product line domain model (i.e., the creation of commonality and variability). Once variability exists, a product derivation can begin, in a tool that implements the SPREBA approach. In this tool, starting a product derivation enables the *Decision* column in the decision table and activates the dynamic weaver—see the right-hand side of Figure 8.2. As soon as a product derivation has started, the tool awaits a user's variability binding decision, see the left-hand side of Figure 8.2. Once a variability binding decision has manually been taken, this decision is processed fully automatically. When an engineer decides that the product derivation is complete, an application product specification is already created, which may be reviewed by different stakeholders and serve as input for the final application requirements engineering phase. This final application of requirements engineering then continues with adding additional requirement specifications, that were not yet part of the product line's reusable artifacts, and with fine-tuning the otherwise generated specification. The actual product development (the programming, testing and deployment) may further unveil other necessary refinements of the specification, which should also be included during this phase. Once the product development is complete, a scoping phase may be performed, which adds these refinements and new requirements specifications into the product line domain model as updates and as further reusable artifacts. These should be added in a disciplined manner and be encouraged by the overall process and used tools, to provide a better-quality specification for any future product line activities. Figure 8.2 shows this

general process and highlights the areas relevant for product derivation. Support for application engineering and scoping activities are not in the focus of this thesis. The crucial behavior for a tool to support this process is the required behavior to automatically process a variability binding decision.

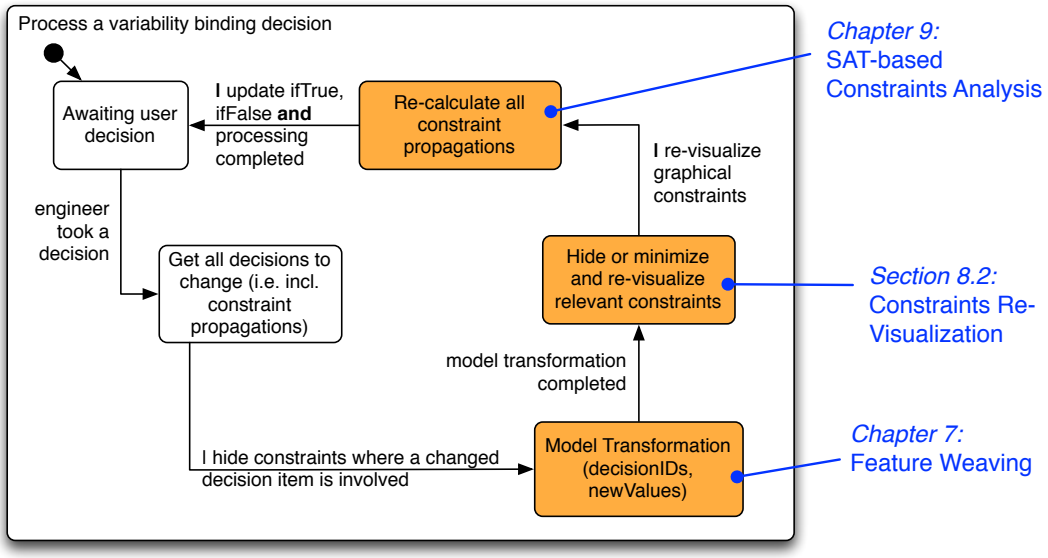


Figure 8.3: A specification of a tool's required behavior to automatically process a manually taken variability binding decision as one step in a stepwise, incremental product derivation.

processing a variability binding decision

Figure 8.3 shows the required behavior of a tool to process a single variability binding decision and, consequently, to implement SPREBA's stepwise, incremental product derivation. In SPREBA the values for the columns *ifTrue* and *ifFalse* are typically already calculated before an actual decision is taken (e.g., recall Figure 6.8) and an engineer can start reasoning and taking a first variability binding decision right away. Whenever a variability binding decision is taken, the tool automatically performs a number of operations, as shown in Figure 8.3. First, along with the just taken variability binding decision, all other decisions required as a constraint propagation are set, to assure a satisfiable configuration. If there are constraints among these decision items, then these constraints will be hidden from the graphic model, before the model can be transformed. Hiding these constraints is necessary because their visualization is only correct for the previous variability configuration and not the subsequent one. Then, the actual *feature weaving* is performed in the state *Model Transformation*, as shown in Figure 8.3. The feature weaving for the manually taken decision and its constraint propagations is specified as a model transformation for all decision items that changed in this step. How this model transformation needs to be executed has already been defined in Chapter 7 (see Figure 7.2). After the feature weaving has been performed, the graphic requirements model is again fully consistent

with the new variability configuration, except for the variability constraints visualization, which is hidden in this state. The previously visualized variability constraints then are derived (or generalized) and re-visualized again, as depicted by the state *Hide or minimize and re-visualize relevant constraints* in Figure 8.3. How this constraint derivation and re-visualization needs to be done is illustrated and specified in Section 8.2. Finally, a fully consistent graphic view on the new configuration is reached and the required constraint propagations in the columns *ifTrue*, *ifFalse*, and *ifUndecided* need to be updated in the decision table view. These updates are calculated fully automatically by evaluating the satisfiability for all particular configurations of interest, as we will show in Chapter 9.
As a result, the original variability binding decision taken by an engineer is fully processed by this behavior, as specified in Figure 8.3. The tool that implements SPREBA generates a consistent and accurate model that fully adequately and consistently visualizes the requirements model and the newly required propagations for the new variability configuration. This functionality is the key to realizing stepwise, incremental product derivation. After such a single variability binding decision has been processed, the SPREBA approach awaits the next variability binding decision from the engineers, as shown in Figure 8.2. This process continues until the engineers decide that their ideally desired product configuration has been reached and the application engineering phase can be started on this basis.

8.2 Deriving and Re-visualizing Variability Constraints

constraints during product derivation

During domain engineering, when all variability decisions are unbound (i.e., set $undecided$), the variability constraints can fully be visualized as introduced in Section 6.1.2. When a decision is bound (i.e., set $true$ or $false$), the constraints where this decision item is involved can not be fully visualized anymore because these join relationships are either woven and/or hidden in the graphic model. SPREBA always propagates the other implied decisions such that the configuration's satisfiability is maintained, to avoid any constraint violations. Therefore, when all decision items involved in a constraint are bound, the constraint is de-facto satisfied and does not need to be visualized anymore. When not all, but only some of the involved decision items are bound, then there are two cases of how to deal with the constraint. In the first case, the already taken decision(s) imply that the constraint is already satisfied, independently of how the remaining involved decisions are taken. Hence, the constraint poses no restriction for the further yet unbound decision items anymore and must not be visualized anymore. In the second case, the already taken decision(s) do not violate the constraint, but the constraint still poses a restriction upon the remaining involved and unbound decision items. This requires that the constraint must be minimized and re-visualized in a derived form, after the feature weaving. Only then an accurate, reduced, but yet consistent product line model can still be visualized.

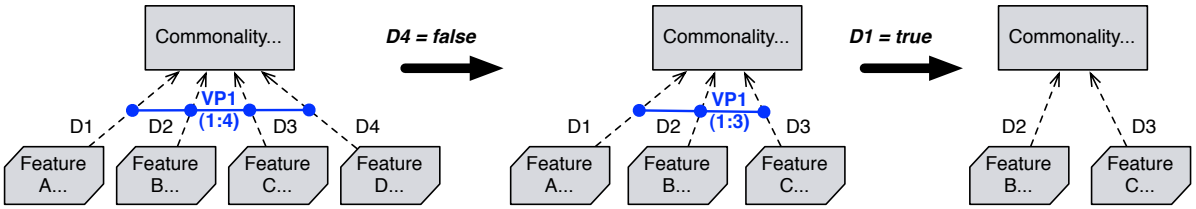


Figure 8.4: An example variation point (VP) constraint and its minimization and hiding after some of its involved decision items are bound.

VP constraint example

Figure 8.4 shows an example variation point (VP) constraint that defines a logical OR relationship among the decision items D1-D4 (i.e., at least one of them must be chosen). If one of these four decision items is deselected, then the constraint still poses a restriction among the remaining decision items because its lower-bound is not yet satisfied. The diagram in the middle of Figure 8.4 shows such a configuration where D4 was deselected and the constraint still remains as a logical OR relationship among the remaining three decision items. If one of these decision items is selected, however, the minimum cardinality of the VP constraint is satisfied. The maximum cardinality is in any case also satisfied because it only provides an upper-bound restriction and correlates with the number of involved decision items in this example. The diagram on the right-hand side of Figure 8.4 shows the model after decision item D1 was selected as a further decision. This reduced the VP constraint to the cardinalities $(0{:}2)$, which allows any arbitrary selection and deselection among the remaining two decision items. Thus, the VP constraint is actually satisfied and poses no further restriction among the remaining two decision items. Hence, the VP constraint is not visualized anymore in this partial configuration.

C constraint example

Figure 8.5 further shows an example arbitrary constraint (C) in Boolean algebra, see C1 in the *Constraints Table*. The constraint specifies that whenever either D1 or D2 or both are selected, then D3 must be deselected and D4 must be selected. The left-most diagram in Figure 8.5 shows the full visualization of this constraint. The diagram in the middle further shows a reduced visualization of the constraint, where decision item D4 has been selected. The selection of D4 already partially satisfies the consequent required by this constraint, but does not fully satisfy it. Whenever either D1 or D2 are chosen, this still requires the deselection of D3. Exactly this constraint is now visualized. Note that the arrow that visualizes the implication has turned from green to red color because the antecedent in this reduced version of the constraint is fully negated and the implication, thus, defines a negative relationship, in red color, recall Figure 6.7. The other way around, when D3 would be selected, this would violate the consequent of the constraint and require that the antecedent must also be $false$ (i.e., that both D1 and D2 must be set $false$). The top-right diagram in Figure 8.5 further shows an even more reduced version of the constraint, where also decision item D1 was deselected. This deselection still leaves D2 open as a remaining condition for the antecedent of constraint C1. The now displayed remaining constraint is merely a simple *excludes* dependency, as also known from feature diagrams or

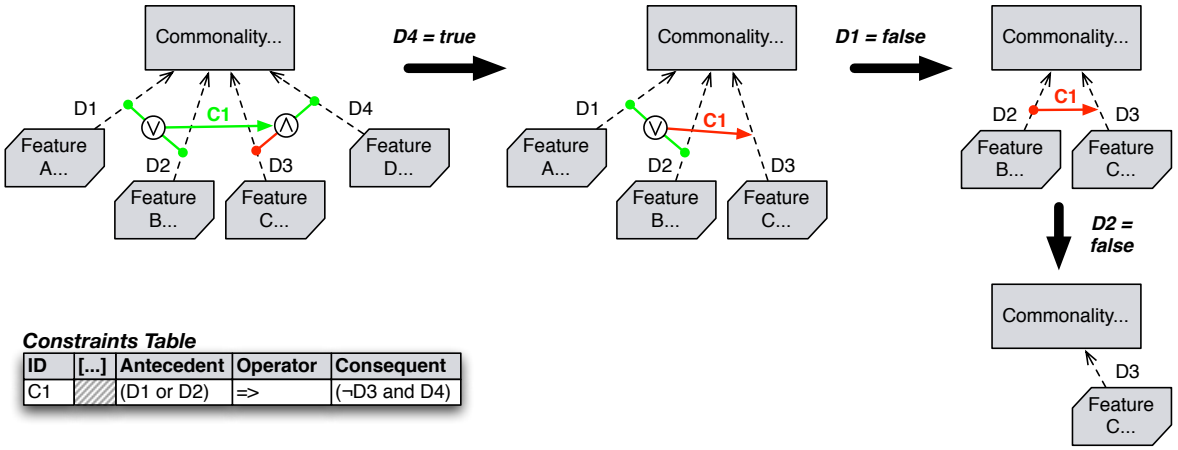


Figure 8.5: An example arbitrary variability constraint in Boolean algebra (C) and its minimization and hiding after some of its involved decision items are bound.

OVM diagrams. Finally, the last diagram in the bottom right of Figure 8.5 shows a configuration where also D2 has been deselected. Having both D1 and D2 deselected renders the constraint's antecedent $false$ and, thus, does not require any restriction among the decision items involved in the consequent anymore. Therefore, D3 remains as a purely optional feature in this last diagram and the constraint C1 is satisfied and not visualized anymore.

deriving variability constraints

Every variability constraint that has one or more involved decision items bound either needs to be hidden, in case it is already satisfied, or needs to be visualized in a derived form, in case it still yields a restriction on the remaining unbound decision items. Decisions can also manually be changed from bound back to unbound during a product derivation, by a user. Therefore, a tool supported solution must also support generalizing already derived constraints. The generalization of already reduced or hidden constraints (i.e., when a bound decision item is set back to $undecided$) is performed indirectly, by taking the constraint's complete form and re-processing (i.e., minimizing or hiding) it.

Figure 8.6 shows this required behavior to minimize or hide all involved visual constraints for a set of decision item changes. Decision items are denoted as Dx in Figure 8.6. This behavior first identifies all VP and C constraints to process. These are all those for which the truth value of one of the involved decision items changed. Further, it derives C and VP constraint visualizations as follows.

A VP constraint is hidden when it is already satisfied (i.e., when its required min and max cardinalities are already reached or when all its involved decision items are already bound), see the top-right part of Figure 8.6. Or it is reduced and visualized on the remaining unbound decision items only, when it is not yet satisfied. A reduced VP constraint also requires adapted cardinality values, to ensure consistency. This simply requires adapting the $minCard$ and/or $maxCard$

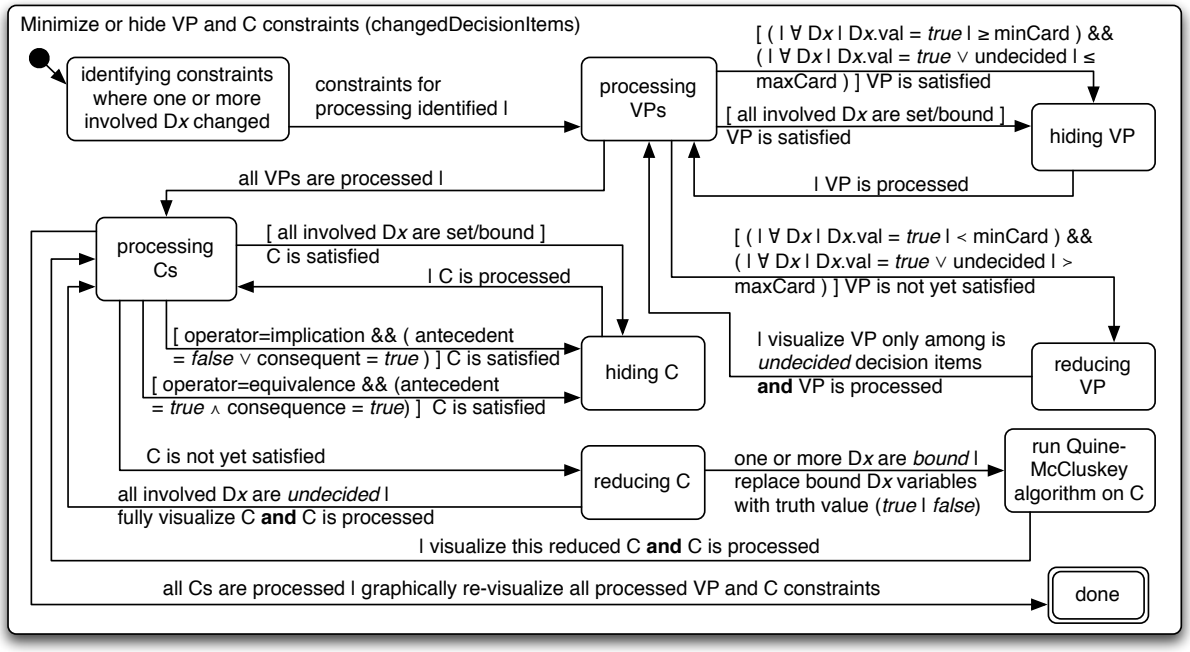


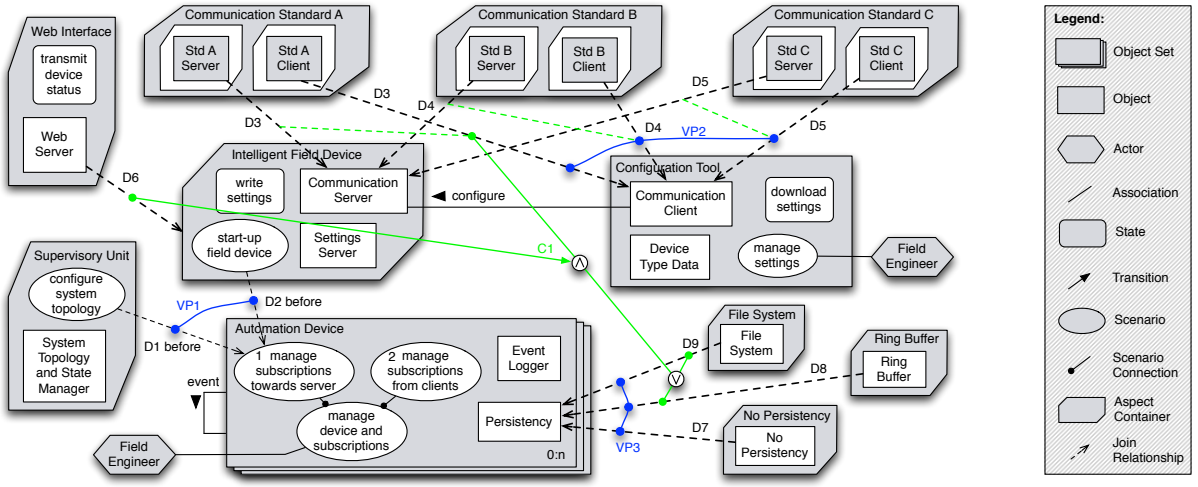
Figure 8.6: A detailed specification of the required behavior for either hiding or minimizing variability constraints during a stepwise, incremental product derivation.

values, see the middle diagram of Figure 8.4, for example. After all VPs are adequately reduced and/or hidden, the impacted C constraints are processed.

As shown in Figure 8.6, a C constraint is satisfied when either all its involved decision items are bound, when its antecedent term is already satisfied (i.e., in case the used operator is an *implication*), or when both its antecedent and its consequent term are satisfied (i.e., in case the operator is an *equivalence*). In these cases the C constraint is hidden. When all these conditions do not apply, however, then the C constraint is not yet satisfied and needs to be reduced. Further, when all its involved decision items are unbound (i.e., set $undecided$), then the C constraint is fully visualized. When some of them are bound, then the C constraint is reduced by using an existing algorithm for the minimization of Boolean formulas. Out of several possible algorithms we chose the Quine-McCluskey algorithm, see [McCluskey, 1956], which is well known in the computer hardware industry for simplifying digital circuits. This algorithm is simple to implement and always finds a minimized form of the given constraint. The only disadvantage is that it computes only one minimized form of the constraint, even when there are several. It could be that a different but equally minimal form of the constraint would be more intuitive to display in the graphical model. This, however, did not turn out as a considerable limitation yet, since most C constraints are not that large and complex. When all C constraints are processed

as well, the derivation of variability constraints is complete and all VP and C constraints can be re-visualized in the graphic model, in their derived form.

8.3 Illustration by a Real-World Example



Decision Table

ID	Description / Derivation Question	Design Rationale	Constraints	ifTrue	ifFalse	Decision
D1	Should the automation device be a supervisory unit?	The autom. dev. platform is used for cost and quality reasons.	VP1	¬D2 ¬D6	D2	*undecided*
D2	Should the automation device be an intelligent field device?	The autom. dev. platform is used for cost and quality reasons.	VP1	¬D1	D1 ¬D6	*undecided*
D3	Should the intelligent field device support communic. std. A?	Standard A is the newest and most performant one.	VP2, C1		¬D6	*undecided*
D4	Should the intelligent field device support communic. std. B?	Is a must in some countries.	VP2			*undecided*
D5	Should the intelligent field device support communic. std. C?	C is still required by many legacy systems.	VP2			*undecided*
D6	Should the intelligent field device a web interface?	The web interface allows access over the world wide web.	C1	¬D1 D2 D3 ¬D7		*undecided*
D7	Is there no memory in the automation device?	Might be interesting for non-critical systems.	VP3	¬D6 ¬D8 ¬D9		*undecided*
D8	Is there a ring buffer in the automation device?	Ring buffers are cheap to implement and fast.	VP3, C1	¬D7 ¬D9		*undecided*
D9	Is there a file system in the automation device?	File system offer huge amounts of storage space.	VP3, C1	¬D7 ¬D8		*undecided*

Variation Points Table

ID	Description / Rationale	minCard	maxCard	Decisions Involved
VP1	An automation device always can be either a supervisory unit or an intelligend field device.	1	1	D1, D2
VP2	There are three different communication standards to configure a field device: standard A, B, and C. At least one must be chosen.	1	3	D3, D4, D5
VP3	There exist three different persistency types: no persistency, a ring buffer, or a file system. One of these must be selected.	1	1	D7, D8, D9

Constraints Table

ID	Description / Rationale	Antecedent	Operator	Consequent
C1	A web interface always requires the communication standard A and a local persistency installed (either ring buffer or FS).	D6	=>	D3 and (D8 or D9)

Figure 8.7: An automation device product line specified in the ADORA language, as in [Stoiber and Glinz, 2009].

This section briefly demonstrates stepwise, incremental product derivation by example. A similar illustration has already been shown in [Stoiber and Glinz, 2010b, Fig. 4]. This section, however, does not only show a straight-forward derivation, where only unbound decisions are bound, as depicted in Figure 8.1, for example, but also includes a case where an automatically propagated decision is changed by the user. Such a manual change of a propagated decision introduces a conflict and, hence, causes unsatisfiability by definition. SPREBA's SAT-based

automated analysis solution, however, is also capable of automatically resolving such conflicts, by finding a minimal set of changes to all other decision items, such that the new configuration is again satisfiable. This is a novel contribution of this thesis. Otherwise, this illustration is similar to what was already presented in [Stoiber and Glinz, 2010b].

real-world exemplar

We chose the real-world industrial software product line exemplar that we already presented in [Stoiber and Glinz, 2009]. Figure 8.7 shows the full ADORA and SPREBA model of this exemplar. The model specifies the software of smart devices for controlling the electricity in power networks in a quite abstract form. These devices are either field devices that directly operate in electrical grids, or supervisory units, which are used to steer the behavior of field devices. The requirements for configuration tools to configure field devices are also modeled. More background can be found in [Stoiber and Glinz, 2009]. We chose to use this example because it is a very abstract model at about the right size to demonstrate both feature weaving and constraint propagation.
On a side-note, note that the presented example always visualizes all variability constraints. This is to better illustrate how variability constraints are derived along with feature weaving. For larger examples, such a complete visualization of all variability constraints will most likely not scale well. Hence, we suggest a selective visualization of variability constraints—as introduced in [Stoiber et al., 2008]—in tools that realize SPREBA.

illustration

Figures 8.8 and 8.9 illustrate an example stepwise, incremental product derivation with eight steps, until a final product specification and configuration is reached. In every diagram one particular variability binding decision is taken, which is highlighted in yellow color in the decision table. While the two Figures should be rather self-explanatory, the following highlights some important details shown. Diagram 1 in Figure 8.8 shows how the feature *Intelligent Field Device* was selected and woven and how this composition also led to a re-targeting of the join relationships of this feature's child features. Compared to Figure 8.7 the JR associated with decision item D6 now directly targets the object set *Automation Device* and the JRs that previously targeted the *Communication Server* component now target the same component in a different parent container. This constraint propagation (D1 $= false$) caused VP1 to be satisfied and hidden in this view. The selection of D4 $= true$ in diagram 2 further satisfies VP2, which, therefore, is not visualized anymore. The deselection of D8 in diagram 3 further leads to a reduction of two constraints: VP3 and C1—both of them are derived with D8 $= false$. In diagram 4, D6 is selected and this leads to two further selections as required by constraint C1. Transitively, this also led to a deselection of D7 because of the constraint VP3. Diagram 5 in Figure 8.9 now introduces a manual conflict introduced by an engineer—the already automatically set decision item D9, as propagated to satisfy constraint C1, is manually set back to $undecided$. This reversion still does not require a reversion of the selection of D6, but it requires setting decision item D8 back to $undecided$. Otherwise, D9 could not validly be set $false$, such that all constraints (i.e., C1 in this case) are still satisfied. Recall, that for an $undecided$ decision it must be satisfiable to set it $true$ *and* $false$ by only propagating yet unbound decision items,

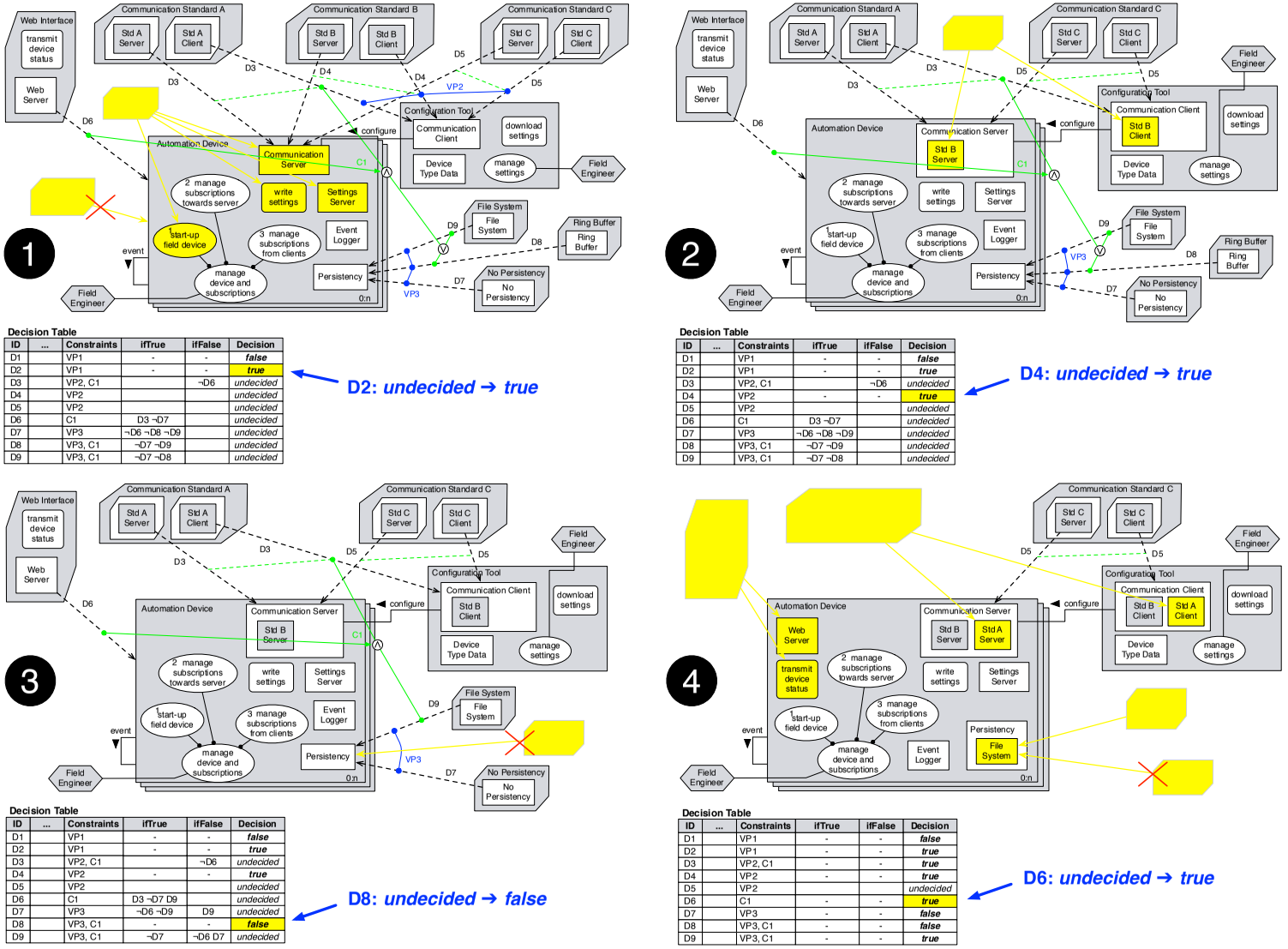


Figure 8.8: An example stepwise, incremental product derivation illustrated on a real-world product line exemplar; compared to [Stoiber and Glinz, 2009] this illustration also includes a manual reversion of an automatically set variability binding decision in step 5; continued in Figure 8.9.

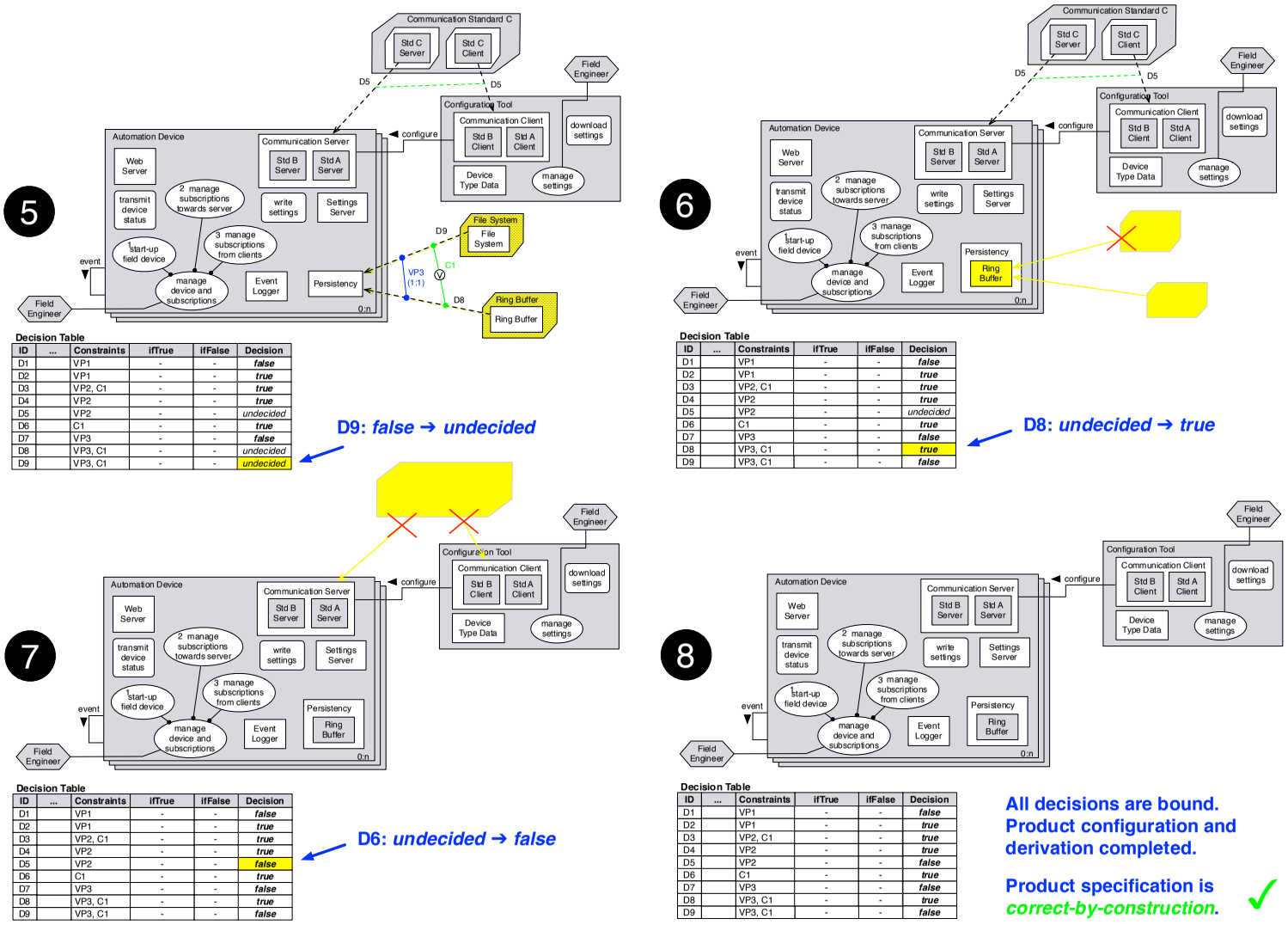


Figure 8.9: An example stepwise, incremental product derivation illustrated on a real-world product line exemplar; compared to [Stoiber and Glinz, 2009] this illustration also includes a manual reversion of an automatically set variability binding decision in step 5; continued from Figure 8.8.

without a need to change any other, bound decision items. Further, diagram 5 also shows the re-visualization of the remaining OR-fragment of the consequent of the constraint C1, which must still hold in this new, generalized configuration. In diagram 6, D8 is selected, which leads to a deselection of D9. In diagram 7, D6 is deselected as a last optional decision. Finally, in diagram 8, a full product configuration and a fully derived product requirements specification is reached, which satisfies all variability constraints. In this manner an engineer can take or change any variability binding decision at any time during a stepwise, incremental product derivation in SPREBA.

automated analysis

The automated analysis that is required to fully automatically process any such selection, deselection, or change for any satisfiable SPREBA model is not trivial. Batory has provided an overview and the fundamentals of how such an automated analysis can be done with feature diagrams [Batory, 2005]. SPREBA models are considerably more complex, however, as they also allow arbitrary cross-cutting and weak hierarchical dependencies. Chapter 9 presents how this automated variability constraints analysis is realized in SPREBA.

CHAPTER 9

SAT-Based Automated Constraints Analysis

The automated analysis of a product line model allows verifying the correctness of the product line domain model and many other crucial operations, such as identifying dead or mandatory decision items and constraint propagation. An *automated* variability constraints analysis is particularly desirable because these tasks are needed but can be very laborious and non-trivial when done manually. Existing approaches already allow many crucial automated analysis operations to support product line engineers. Most of these existing approaches are based on feature models, recall Section 3.3. Because SPREBA models significantly differ from feature models (i.e., they can contain cross-cutting and weak hierarchical dependencies), these solutions can not be used with SPREBA.

correctness-by-construction

In the SPREBA approach, the correctness (i.e., the absence of conflicts) of a product line domain model is continuously verified already at modeling time. This is different from some existing approaches, where inconsistencies are temporarily allowed, e.g., [Trinidad et al., 2008]. When building on SAT solving for the model's automated analysis, such inconsistency can void all automated analysis capabilities. For example, when a constraint $D1 \Rightarrow \neg D1$ is defined, the model as a whole would always be unsatisfiable and any partial assignment of the model would also be unsatisfiable. Hence, our SAT-based analysis, as follows, would not be able to work correctly. Therefore, we introduce an approach that always preserves the *satisfiability* of the model as a top priority. We argue that it is beneficial to resolve any conflicting constraint specification immediately or as early as possible. A postponed and late detection and rectification of an inconsistency may already have propagated the defect to other development phases or

may make the later-on resolution more difficult. Especially when different stakeholders have to fix an inconsistency later, they have to rely on documented design rationales, which generally leads to even larger efforts and risks for misunderstanding—depending on the quality of the documentation. Interestingly, a recent study of configuration challenges among the users of the Linux kernel and eCos configurators has shown that users actually also prefer resolving conflicts immediately [Hubaux et al., 2012]. Hence, SPREBA puts a high priority on verifying the satisfiability of the model as a whole at any time and on immediately notifying a modeler when he or she performs a change that leads to unsatisfiability.

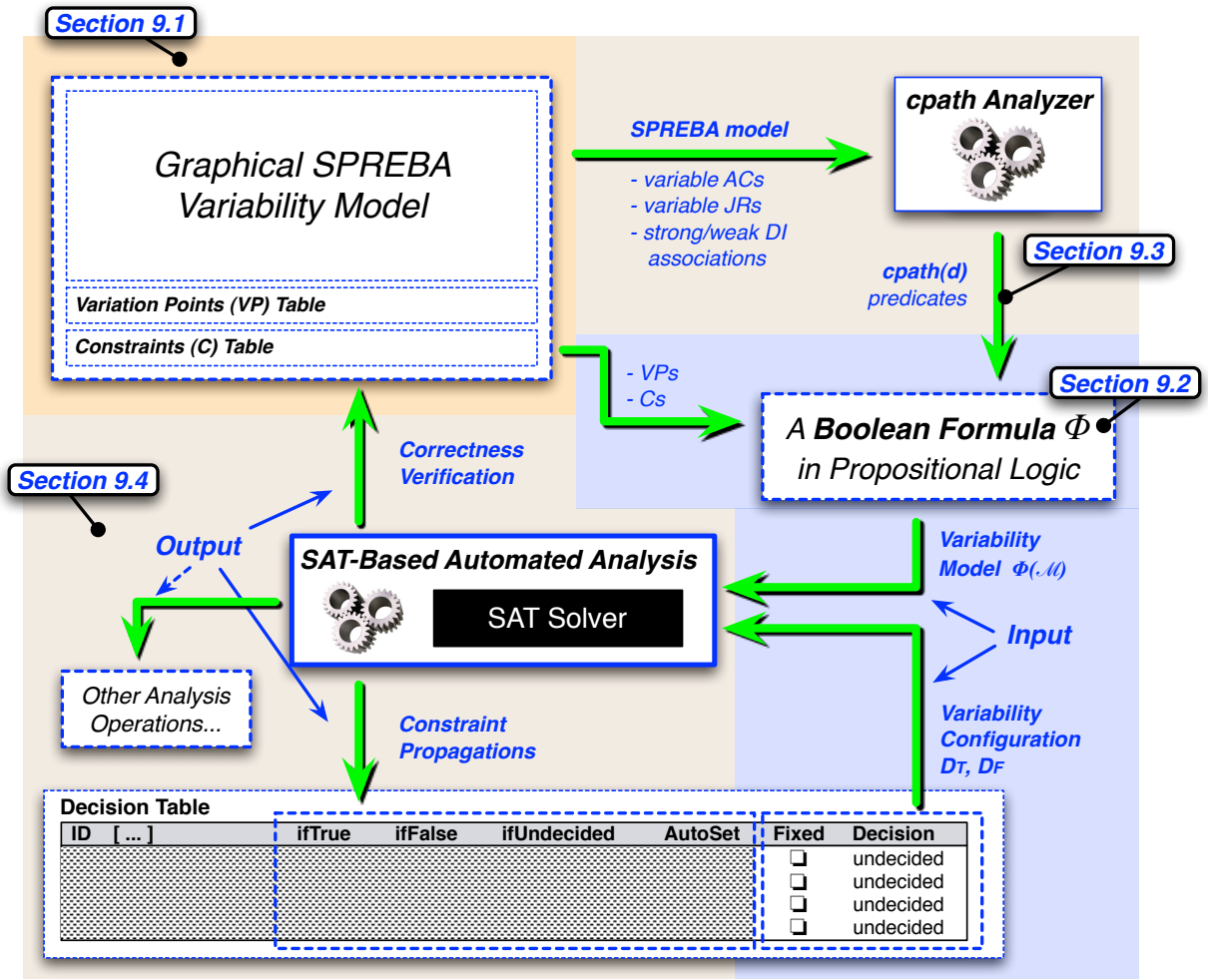


Figure 9.1: An overview of SPREBA's SAT-based automated variability analysis.

overview This chapter introduces SPREBA's SAT-based automated analysis solution. Figure 9.1 provides a complete overview on this chapter, which is decomposed into four sections, as also highlighted in Figure 9.1. First, Section 9.1 provides precise definitions of the formal semantics of

all ADORA model elements that constitute the SPREBA variability model. These definitions provide the basis for an automated parsing of the variability model, as described in Section 9.3—see the top and top-right of Figure 9.1. Before this parsing process is described in detail, Section 9.2 describes how a SPREBA model is defined as an equivalent Boolean formula Φ in propositional logic, which provides the core artifact for all the SAT-based analysis. This Boolean formula Φ includes all variability constraints, their hierarchical dependencies, and the hierarchy constraints for all decision items. The main ingredients for constructing this formula Φ, compared to feature model-based approaches, are the so-called *'commonality path'* predicates, $cpath$ predicates in short, for all decision items in the model. These are formalizations of every decision item's required hierarchy constraint, such that it influences the commonality precisely as specified. Section 9.3 illustrates how the SPREBA model is parsed and how these predicates must be defined for any valid SPREBA model. Finally, Section 9.4 presents SPREBA's automated analysis solution, where Φ is already given as a Boolean logic equivalent to the SPREBA model at hand, see Figure 9.1. Subsection 9.4.1 shows how SPREBA assures that the domain model stays satisfiable at all time and how dead and mandatory decision items can easily be identified with a SAT solver. Subsection 9.4.2 shows how the complete set of constraint propagations is calculated for any new decision or any change of an already bound decision. This solution takes the already set decisions into account and always provides a minimal set of required propagations or changes to already set decision items, such that the resulting new configuration allows any still undecided decision item to be set either way and to still reach a full product configuration that satisfies all constraints.

9.1 Variability-relevant Model Elements

A verifiably correct translation of a SPREBA variability model into a Boolean formula requires a precise definition of all relevant model elements from which this formula is created. When realizing SPREBA with the ADORA modeling language, many types of model elements do not at all or only indirectly influence the actual variability model. Examples for such model elements would be *abstract objects*, *states*, or *scenarios*, which are crucial for the underlying ADORA requirements model, but not for the inherent variability model. They can at most indirectly influence the variability model by their nesting in other model elements, when they are associated with *variable join relationships*. On the other hand, the model elements that are crucial and that constitute the actual SPREBA variability model are, for example, *variable aspect containers*, *variable join relationships*, *decision items*, and *VP* and *C constraints*. These were already introduced on a more abstract level in the Chapters 5 and 6. The following provides a more precise and partially formal definition of those elements that need to be interpreted to derive the formula Φ—the logical equivalent of the SPREBA variability model. These explanations shall provide a sound and unambiguous basis for a formal interpretation.

Aspect container (AC). Aspect containers are those model elements that contain diagrams or fragments of diagrams that can be woven into the core model or into other aspects. ADORA aspects are either used for a non-redundant specification of cross-cutting concerns [Meier, 2009], or for specifying variable features, which do not necessarily need to be cross-cutting.
Formally, an aspect container a is defined as an element (C, p, b) from the set $\mathcal{P}(M) \times M \times \{true, false\}$. The first component C is the set of all nested or child model elements c within this aspect container, where C can be any combination of model elements of the set of all model elements M, which yields a valid model or model fragment in ADORA [Meier, 2009]. The second component p is the parent model element in which this aspect container a is contained (this can be another aspect container or an abstract object, as far as [Meier, 2009]'s constraints allow, or the ADORA model $\mathcal{M}$ itself, if not nested anywhere). The third component b eventually determines whether or not a is variable (i.e., $false$ defines a is a plain cross-cutting concern and $true$ defines it to model variability). Whenever an aspect container is defined as variable (i.e., $b = true$) this requires an annotation of a decision item d on every outgoing join relationship j of this aspect. The term *aspect* is used to reference an AC and all its outgoing JRs.
A variable aspect containers (with $b = true$) is either a 'feature' or 'part of a feature' (i.e., of a heterogeneously cross-cutting variable feature). This distinction is not relevant for parsing a SPREBA model into a Boolean formula, but it certainly is for creating a well-structured requirements specification from a modeler's point of view. Whenever a variable aspect container is *not* nested within any other variable aspect container, then it is a *feature*. Whenever it is nested in another variable aspect container, then it is a *part of a feature*.

Join relationship (JR). Join relationships are those model elements that determine where and how the contents of an aspect container are composed (i.e., the join points and weaving types). The weaving types of join relationships do not have any influence on the actual SPREBA variability model. The source and target model elements of join relationships (the latter correspond to *join points*) are crucial for the calculation of $cpath$ predicates. Outgoing join relationships of a variable feature or part of a feature lead to hierarchical dependencies. When a join relationship associated with a particular decision item d targets another variable AC, then there is a hierarchical variability dependency—no matter what weaving type is specified (i.e., whether the targeted model element will be extended with *before* or *after* or replaced with *instead* [Meier, 2009]). Variable JRs lead to hierarchical variability dependencies in a Boolean way: wherever a variable join relationship targets another variability, there is a functional dependency, and where it does not, there is none.
Formally, a join relationship j is an element (s, t, d, w) from the set $M \times M \times (D \cup \{\emptyset\}) \times \{true, false\}$. The first and second components s and t are the source and target model elements, which must yield a valid and composable aspect-oriented model. For example, a j connecting a state towards a scenario is neither valid nor composable, while one connecting a state towards a state transition is. These well-formedness constraints are defined in [Meier, 2009] and were slightly generalized in [Kandrical, 2009] and [Jehle, 2010]. The third component d contains the decision item that is associated with this join relationship. Every variable

join relationship (i.e., that has its source node s nested within an aspect a with $b(a) = true$) must have exactly one decision item d associated with it. When the JR's source node s is nested within an a that is not variable (i.e., $b(a) = false$), then the component $d(j)$ must be the empty set. The fourth component w eventually specifies whether the decision item is weakly associated with this join relationship or not (*true* stands for weak and *false* for a strong association). Strong association requires that the element s must eventually impact the commonality of the variability model when d is decided to $true$ (i.e., a classic child-parent dependency). Weak association, in contrast, does not necessarily require an impact of this j on the commonality, when d is set $true$. These dependencies will be formalized into $cpath$ predicates, as follows in Section 9.3.

Decision item (DI). The core element that constitutes a SPREBA variability model in its Boolean form is the *decision item*. Decision items are those model elements that actually trigger a feature weaving when selected, deselected, or changed (recall Chapter 7). A decision item is uniquely identifiable and always associated with at least one variable join relationship in the SPREBA domain model.
Formally, a decision item d is an element (J, v) from the set $\mathcal{P}(J) \times \{true, false, undecided\}$. The first component J is the set of all join relationships this decision item is associated with, which must not be the empty set (a decision item that isn't associated to any JR will be deleted). The second component v is the decision item's truth value, which is always one of the values $\{true, false, undecided\}$. This truth value determines how all associated join relationships J and their involved ACs need to be visualized in the graphic model—recall Chapter 7.

Variation point (VP) constraints. VP constraints are cardinality-based variability constraints—please refer to the Sections 6.1.2 and 8.2 for a general introduction.
Formally, a VP constraint v is an element (D, n, m) from the set $\mathcal{P}(D) \times \mathbb{N} \times \mathbb{N}$, where $\mathcal{P}(D)$ is the family of sets over all decision items. The first component D is the set of involved decision items. The second component n is the minimal number of decision items in D that need to be set $true$. The third component m is the maximum number of decision items in D that may be set $true$. Every VP constraint yields two predicates that have to be satisfied, which are the predicates $min(D(v), n)$ and $max(D(v), m)$, where $D(v)$ is set of involved decision items and n and m are natural numbers. Thus, the constraint yielded by the second component n can be formalized as $min(D(v), n) = \bigvee_{D \in \binom{D(v)}{n}} \bigwedge_{d \in D} d$. The constraint yielded by the third component m can consequently be formalized as $max(D(v), m) = \neg min(D(v), m + 1)$. Such a constraint specification can straightforwardly be translated into a Boolean formula, which exhaustively lists disjunctions of conjunctions of all valid configurations that satisfy the VP's min and max predicates (recall the example VP constraint « $\{D1, D2, D3\}, 1, 1$ » in Section 6.1.2, for example).

Arbitrary Boolean (C) constraints. Also C constraints can be defined as arbitrary Boolean formulae in propositional logic—see the Sections 6.1.2 and 8.2 for a general introduction.

Formally, a constraint c is an element (t_a, o, t_c), which uses the following grammar:

$$
\begin{aligned}
&c = t_a \; o \; t_c \\
&t_a = T \\
&t_c = T \\
&o = \text{"}\Rightarrow\text{"} \;\mid\; \text{"}\Leftrightarrow\text{"} \\
&T = \text{"}true\text{"} \mid \text{"}false\text{"} \mid \text{"}D\text{"}\mathbb{N} \mid T \text{ "}\vee\text{" } T \mid T \text{ "}\wedge\text{" } T \mid\; T \text{ "}\oplus\text{" } T \mid\; \text{"}\neg\text{"T}
\end{aligned}
$$

A JavaCC[1]-generated parser is then used to handle these c constraints. The *antecedent* can be any arbitrary term T and is always to the left of the *operator*, which is typically an implication ($\Rightarrow$), but can also be an equivalence ($\Leftrightarrow$). The *consequent* is also an arbitrary term T. Note that for an equivalence both terms become antecedent and consequent, as this yields a bidirectional implication.

Commonality. The commonality in a SPREBA model are all model elements that are not contained in variable aspect containers. Every other model element that is not itself a feature (or a part of a feature), contained in a feature (or in a part of a feature), a variable join relationship, or a decision item, is part of the commonality.

Conventional aspects. ADORA's conventional aspect modeling concept [Meier, 2009] is fully orthogonal to the SPREBA concepts. Conventional aspects are handled like any other ADORA model element. Whether conventional aspects are visualized in an aspect-oriented or in the woven view does not influence the SPREBA model in any way. The visualization of conventional aspects can be switched between the woven or aspect-oriented view at all times, even during a product derivation—ADORA's aspect weaver has been evolved this way since [Kandrical, 2009], as already presented in Chapter 7.

no aspects without JRs

Restrictions for Features. Variable ACs are automatically recognized as *features* or *parts of a feature*. Every 'feature' or 'part of a feature' is only valid when it has at least one outgoing join relationship. Otherwise, this variable aspect can not have any impact on the remaining model and is void. This constraint is required in addition to Meier's language constraints [Meier, 2009].

no scattering of DIs over features

Further, to keep the variability in a SPREBA model reasonably simple to comprehend for human engineers, we additionally defined the restriction that decision items are *not* allowed to be scattered over multiple features. Engineers typically associate features with choices. In SPREBA, the actual configuration is done with decision items and not features, however. This allows a more fine-grained variability modeling, wherever necessary. As specified so far, decision items could be arbitrarily associated with the variable join relationships (provided that every variable join relationship has exactly one decision item associated with it). When one decision item is associated with join relationships of multiple features, this could complicate the comprehension of the impact of such a variability binding decisions profoundly. In fact, if such a case occurs,

[1] See http://javacc.java.net/ (checked on July 1, 2012).

we advise a refactoring that merges all these variable model elements into a single feature, as they can anyway only vary together. To avoid such a decision item scattering we introduce the restriction that every decision item can only be associated with join relationships of at most one variable feature. Vice versa, however, arbitrarily many decision items can still be used among the join relationships of a specific feature (i.e., where none of them is allowed to be reused with any JR of another feature). Formally, this constraint can be defined as follows. When F is the set of all features and D_f is the set of decision items associated with the outgoing JRs of f and all its nested parts, then $(f, f') \in F \times F \mid D_f \cap D_{f'} = \emptyset$ must always hold.

9.2 SPREBA Models as SAT Problems

In the Boolean satisfiability problem (SAT), one is given a Boolean formula in conjunctive normal and determines whether there exists an assignment that satisfies all the clauses. If such an assignment exists, the formula is called *satisfiable*. Otherwise, it is called *unsatisfiable*. The SAT problem is NP-complete and was in fact the first known NP-complete problem [Cook, 1971]. The fact that the problem is NP-complete means that it is not possible to guarantee that an answer can be found in polynomial time on any formula, unless $P = NP$. Despite this theoretical lower bound, there still exist many practical instances for which satisfiability can be decided very fast. These practical instances include feature models and SPREBA models, as Chapter 13 shows.

Our principal task is to create a Boolean formula in propositional logic that is *equivalent* to the SPREBA variability model that is analyzed. Equivalent means that the formula corresponds exactly to the formal semantics of the variability model (i.e., its hierarchies, cross-cutting, and constraints, see Section 9.1). Such a Boolean formula allows us to perform an automated analysis of the variability model, based on a SAT-solver. The further translation of such Boolean formulae into a CNF format is straightforward. In the following a similar notation as in [Welzl, 2005] is used. The given ADORA and SPREBA model as a whole is denoted as $\mathcal{M}$ and its equivalent Boolean formula, which is the formal equivalent of the actual SPREBA variability model, is denoted by $\Phi(\mathcal{M})$.

a SPREBA model as a SAT problem

Equation 9.1 shows the general form of a SPREBA model as a SAT problem. For a given model $\mathcal{M}$, $V(\mathcal{M})$ denotes the set of all variation point constraints, $C(\mathcal{M})$ the set of all other logical constraints and $D(\mathcal{M})$ the set of all the decision items.

$$\Phi(\mathcal{M}) := \underbrace{\left(\bigwedge_{v \in V(\mathcal{M})} t_{vp}(v) \right)}_{(1)} \wedge \underbrace{\left(\bigwedge_{c \in C(\mathcal{M})} t_{con}(c) \right)}_{(2)} \wedge \underbrace{\left(\bigwedge_{d \in D(\mathcal{M})} v(d) \Rightarrow cpath(d) \right)}_{(3)} \tag{9.1}$$

The key components in this formula are (1) the conditions $t_{vp}(vp)$ that need to hold for every VP constraint (its cardinalities and its hierarchy constraint), (2) the conditions $t_c(c)$ that need to hold for every C constraint (its Boolean logic constraint and hierarchy constraint), and (3) that for every decision item its hierarchy constraint $cpath(d)$ must be satisfied when its truth value $v(d)$ is set $true$. The implication in (3) also defines that whenever a decision item's truth value $v(d)$ is set $false$ it is irrelevant whether its hierarchy constraint $cpath(d)$ is satisfied or not. When $v(d)$ is set $undecided$, then $cpath(d)$ must still be satisfiable, however. This is the general formula of how a SPREBA model is represented as a SAT problem.

SAT with partial assignments

Compared to this classic use of SAT we are particularly interested in verifying the satisfiability of variability models along with partial assignments in SPREBA, recall Chapter 8. This requires an explicit inclusion of actual bindings of taken variability decisions into the SAT problem, wherever a decision item's truth value $v(d)$ has already been set $true$ or $false$. The set $D(\mathcal{M})$ contains all decision items. The two disjoint subsets $D_T(\mathcal{M})$ and $D_F(\mathcal{M})$ contain all decision items with their truth value set either $true$ (i.e., $(v(d) == true) \Rightarrow d \in D_T$) or $false$ (i.e., $(v(d) == false) \Rightarrow d \in D_F$), respectively. The sets D_T and D_F must not overlap (i.e., $D_T \cap D_F = \emptyset$) and only decision items that exist in $D(\mathcal{M})$ can be part of D_T and D_F (i.e., $(D_T \cup D_F) \in D(\mathcal{M})$).

$$\Phi(\mathcal{M})_{withAssignm} := \underbrace{\left(\bigwedge_{v \in V(\mathcal{M})} t_{vp}(v)\right)}_{(1)} \wedge \underbrace{\left(\bigwedge_{c \in C(\mathcal{M})} t_{con}(c)\right)}_{(2)} \wedge \underbrace{\left(\bigwedge_{d \in D(\mathcal{M})} v(d) \Rightarrow cpath(d)\right)}_{(3)} \wedge \underbrace{\left(\bigwedge_{d \in D_T(\mathcal{M})} v(d)\right)}_{(4)} \wedge \underbrace{\left(\bigwedge_{d \in D_F(\mathcal{M})} \neg v(d)\right)}_{(5)} \tag{9.2}$$

Equation 9.2 presents the representation of the full SPREBA model plus a particular partial assignment of variability binding decisions (i.e., decision item's truth values) as a SAT problem. The third term (3) still requires that every selected decision item must always have a satisfied $cpath(d)$ predicate. The remaining two terms include the partial assignment that must be satisfied for the decisions that are already taken. The fourth term (4) enforces that all selected decision items in the set D_T are set $true$, which restricts the remaining variability space of $\Phi(\mathcal{M})$ further and also implies that these decision item's $cpath(d)$ predicates must be satisfied, as defined in the third term. Analogously, the fifth term (5) enforces that all deselected decision items in the set D_F are set $false$. These two terms further restrict $\Phi(\mathcal{M})_{withAssignm}$'s remaining variability space as they fix these values to a specific binding.

Compared to [Welzl, 2005], this handling of partial assignments as additional constraints that are added to the overall formula may seem peculiar because the binding of a variable would

rather correspond to a reduction of the overall formula Φ, there. However, for SPREBA, dealing with bound decision items as additional constraints makes perfect sense, as these decisions are not permanent but can still be changed at any time during the product derivation process. Adding the terms (4) and (5) to Φ essentially has the effect that any overall variable assignment, where a decision in D_T is not set $true$ and a decision in D_F is not set $false$ is *unsatisfiable*. Thus, *satisfiability* of $\Phi(\mathcal{M})_{withAssignm}$ means that there exists a full assignment of all still undecided decision items D_U (where $D_U = D \setminus D_T \setminus D_F$) that satisfies all constraints of the SPREBA model and the already taken decisions (i.e., $D_T \cup D_F$).

SAT solving $\Phi(\mathcal{M})$

A SAT solver essentially returns either $true$ or $false$ as a result. When formula $\Phi(\mathcal{M})$ together with a partial assignment of decision items (D_T, D_F) is verified with a SAT solver, the result $true$ means that the SAT solver has proven that there is a full configuration (i.e., at least one) that satisfies all the constraints. The answer $false$ otherwise means that the SAT solver has proven that all remaining configuration possibilities (i.e., of the decision items not in D_T or D_F) do not satisfy the constraints of the SPREBA model along with this given partial assignment.

an example

A SPREBA model can be translated into an instance Φ (Equation 9.1) and its satisfiability Φ together with a partial assignment (Equation 9.2) can be verified. To demonstrate this a simple example is presented in Figure 9.2. On the top-left of Figure 9.2 a simple SPREBA variability model is shown, which contains only three variable features F1, F2, and F3, where each of them is associated with one decision item, D1, D2, and D3. The features associated with D2 and D3 are child-features of D1, which implies that their $cpath$ predicate requires D1. A VP constraint with cardinalities $0{:}1$ is defined among D2 and D3, which yields a logical OR dependency. This VP constraint also requires D1 to be selected as its $cpath$ predicate; otherwise, it is irrelevant (as all its involved decision items are not part of the product). The detailed calculation of $cpath$ predicates follows in Section 9.3.

The middle part of Figure 9.2 then shows a Boolean formula that precisely specifies this variability model as the formula Φ, as outlined in Equation 9.1. Only the Boolean form of $t_{vp}(\text{VP1})$ is slightly simplified, for brevity.

The bottom part of the Figure eventually shows how the automated evaluation of this formula with a SAT solver works. In the first case, the satisfiability of only Φ is verified, which yields a positive result, i.e., $true$. This result means that the SAT solver has proven that there is at least one configuration of all decision items D1-D3 that satisfies the formula Φ. In the second case, Φ together with a partial assignment of the variability is evaluated—in this case a selection of D1. How this partial assignment needs to be included with the formula Φ is specified in Equation 9.2: D1 becomes part of the set $D_T = \{d_1\}$, while the set D_F still remains empty. As shown in Equation 9.2 all decision items in D_T merely need to be conjunctively added to the term Φ, to evaluate whether there is a configuration that satisfies Φ and that includes D1 as selected (which leads to the logical appendix '$\wedge$ D1'). Similarly, when all decisions are *undecided* and only D2 is set $true$, a mere addition of '$\wedge$ D2' to Φ suffices as well, which is again a satisfiable partial configuration, as shown in Figure 9.2. Finally, in the third case, when D1 is considered

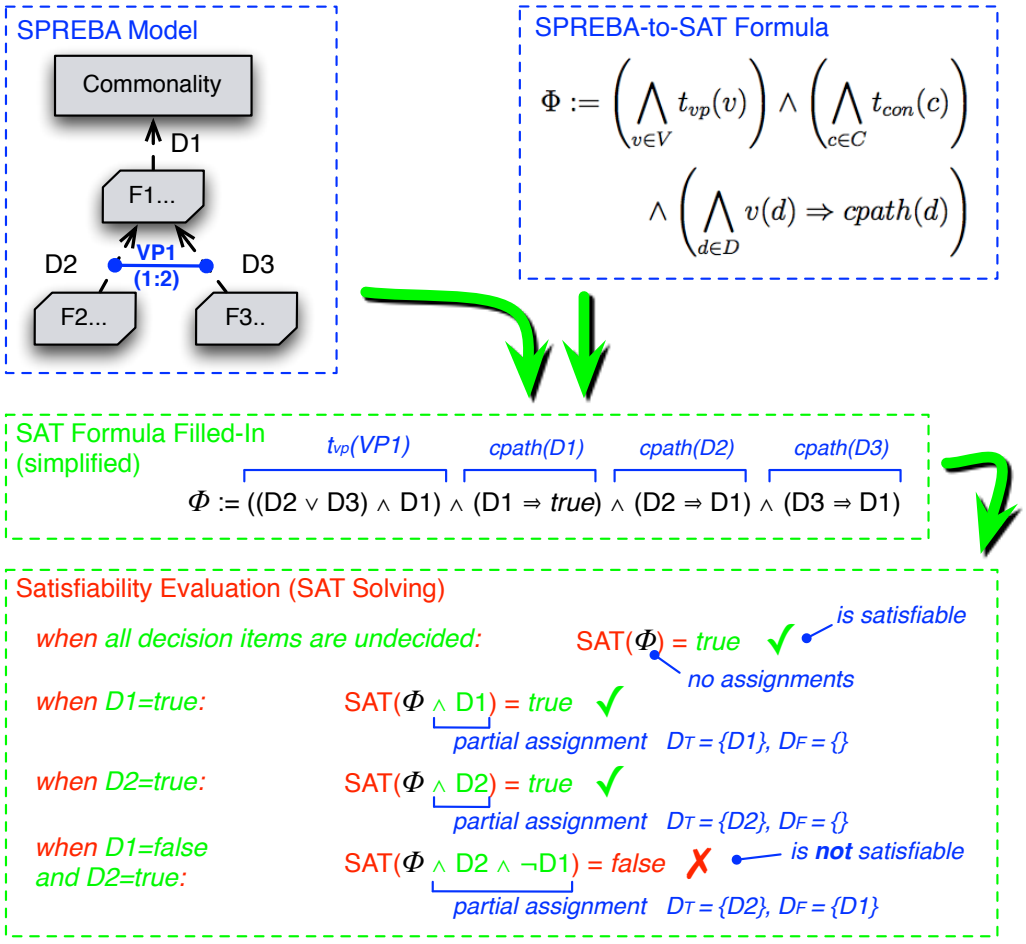


Figure 9.2: An example of how a simple SPREBA model is parsed into an equivalent SAT problem Φ and how its satisfiability with no assignments and with partial assignments is evaluated by SAT solving.

deselected and D2 selected, this yields the sets $D_T = \{d_2\}$ and $D_F = \{d_1\}$, which leads to an extension of Φ with the term '$\wedge$ D2 $\wedge \neg$D1' for which the SAT solver will prove that no satisfiable configuration of all decision items is possible (i.e., no matter how D3 is decided), which consequently yields the result *false*, which stands for *unsatisfiable*. In this particular case the reason for the unsatisfiability is the hierarchy constraint of D2 (see the clause $cpath$(D2): D2 $\Rightarrow$ D1 in the middle part of Figure 9.2), which, as part of Φ, is always unsatisfiable together with '$\wedge$ D2 $\wedge \neg$D1'.

While the automated analysis of SPREBA variability models is rather straightforward, once an equivalent form of the model in Boolean logic has been composed, the main challenge lies in creating this equivalent Boolean formula. In particular, composing the required hierarchy

constraints (i.e., the $cpath$ predicates), when cross-cutting and weakly associated decision items are involved, is non-trivial and requires a recursive solution. The $cpath$ predicate is crucial for formally defining the hierarchy constraint of decision items and for the formal definition of VP and C constraints as well.

$t_{vp}(v)$ predicates

A variation point constraint, as specified in Section 9.1, must only hold when at least one of its involved decision items has a satisfiable $cpath$ predicate. When none of the involved decision items can be set $true$ together with the already existing configuration—without yielding an unsatisfiable configuration—then the VP's specified cardinalities do not need to hold and the VP constraint is also satisfied. Such an example is shown in Figure 9.2, where all involved decision items of VP1 require D1 as their $cpath$ predicate. Thus, when D1 is set $false$, the VP constraint does not need to hold and is satisfied as well. Hence, only when the hierarchy constraint is satisfiable the actually specified cardinalities must also be satisfied for a VP constraint.

When constructing the formula Φ we generate these required t_{vp} predicates as a disjunction between the VP's Boolean form (which typically becomes a quite lengthy term) and the negation of the required hierarchy constraint. This means that whenever the VP's hierarchy constraint is unsatisfiable, the term t_{vp} is satisfied. Otherwise, the required cardinalities must be satisfied. Formally, such variation point constraint predicates t_{vp} are defined as:

$$t_{vp}(D, n, m) := (min(D(v), n) \wedge max(D(v), m)) \vee \left(\bigwedge_{d \in D} \neg cpath(d)\right)$$

Whenever this term evaluates to $true$ the VP constraint is satisfied.

$t_{con}(c)$ predicates

Similarly to VP constraints, also C constraints either must be satisfied or these $cpath$ predicates must be unsatisfiable. However, with C constraints two different cases exist. A C constraint is either based on an implication ($\Rightarrow$) or an equivalence ($\Leftrightarrow$). For an implication the constraint's $cpath$ predicate only needs to include the decision items involved in the $antecedent$ term, as the $consequence$ also depends on this term. When the antecedent (the condition under which the $consequent$ term must hold) is irrelevant, then the constraint as a whole is irrelevant and, hence, satisfied. An equivalence constraint's $cpath$ predicate, on the other hand, needs to include both the antecedent and the consequent term, as the relationship is bi-directional and both terms in fact act as the antecedent for the other term. For an equivalence the $cpath$ predicate must be unsatisfiable for all involved decision items, hence, such that the constraint becomes irrelevant and therefore satisfied.

When constructing the formula Φ we generate the predicate $t_{con}(c)$ as a disjunction of the C's Boolean form and the negation of its required hierarchy constraint, which depends on the used operator $o(c)$. For every C constraint that is defined as an implication (i.e., $o(c) =$ "$\Rightarrow$") the $t_{con}(c)$ predicate is defined as:

$$t_{con}(c) := c \vee \left(\bigwedge_{d \in D(t_a(c))} \neg cpath(d)\right)$$

For every C constraint that is defined as an equivalence (i.e., $o(c) =$ "$\Leftrightarrow$") the $t_{con}(c)$ predicate is defined as:

$$t_{con}(c) := c \vee \left(\bigwedge_{d \in D(c)} \neg cpath(d)\right)$$

Whenever this term $t_{con}(c)$ evaluates to $true$, the C constraint is satisfied.

9.3 The $cpath$ Predicate: Hierarchical Dependencies of Decision Items

state of the art

In today's variability models (e.g., feature models) only strong dependencies between variable features and their parent features exist (i.e., child-parent dependencies). Since every feature has only one parent feature (i.e., in a feature model, except for the root feature) and a strong hierarchical dependency to its parent feature, the formalization of this constraint is straightforward: every feature logically implies its parent feature. Transitively, hence, the conjunction of all child-parent dependencies until the root feature yields the complete hierarchy constraint for a feature.

weak dependencies

In a SPREBA variability model such hierarchy constraints become incomparably more complex. First, variable features (or more precisely, decision items) can have arbitrarily many parent features (parent decision item, respectively). Second, decision items can also be weakly associated with join relationships, which can lead to complex constellations where multiple combinations of paths may satisfy the hierarchy constraint. In fact, these paths can quickly become non-trivial to comprehend for large SPREBA models. However, automatically parsing the hierarchy constraint out of any valid SPREBA model is yet automatable, with a recursive solution.

$cpath$ examples

Figure 9.3 presents a list of examples to illustrate the problem that needs to be solved by the $cpath$ algorithm. Every example includes the necessary $cpath$ predicate for all decision items that do not merely impact the commonality, in which case the $cpath$ predicate is satisfied by default (i.e., because no other decision is required for this functionality to become part of a derived product). The parts of the variability model required for the $cpath$ predicate of the decision item and AC printed in bold are highlighted in blue color in Figure 9.3. Transitively, also all all other variability model elements, that are relevant for the full $cpath$ predicate, are shown. These examples, as presented in Figure 9.3, provide a comprehensive overview of special cases that need to be incorporated in an algorithm that generates the required $cpath$ predicates for any particular decision item in any SPREBA model.

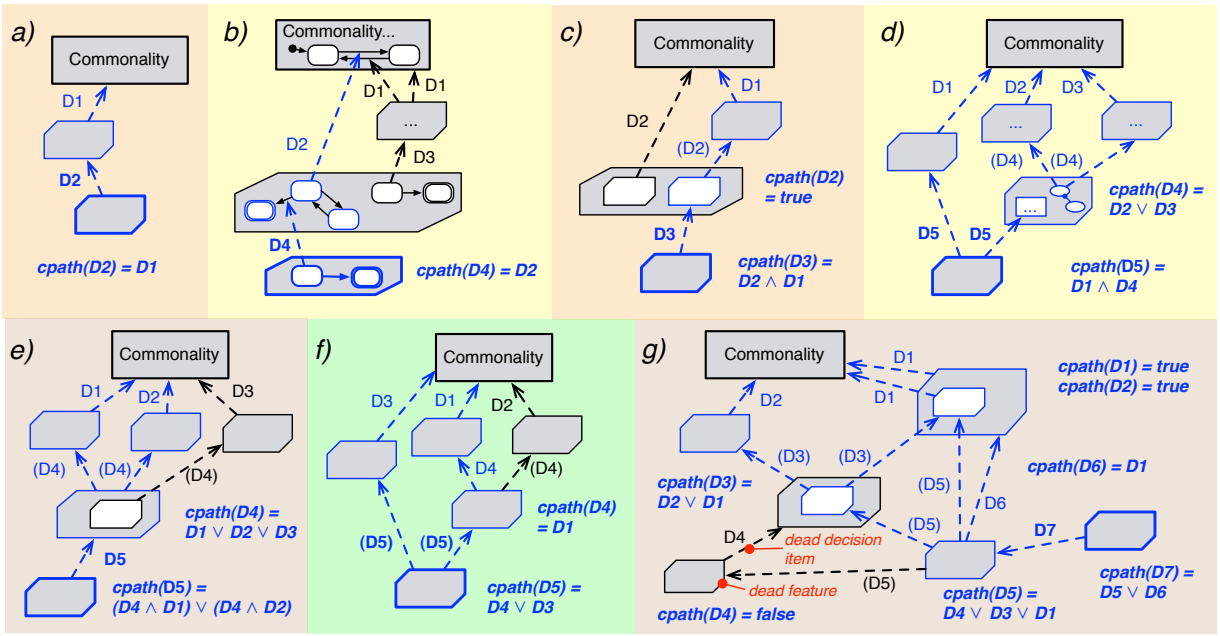


Figure 9.3: Abstract examples of SPREBA models with their weaving paths to the commonality highlighted and the required Boolean $cpath$ predicates shown in blue color.

In diagram $a)$ in Figure 9.3 a trivial case is shown, where D2 is strongly associated with its JR and is a sub-variability of D1. Therefore, the selection of D2 also always requires D1 to be selected, such that the variable model elements introduced with the composition of D2 actually become part of the derived product. This yields the predicate $cpath(\text{D2}) = \text{D1}$.

In diagram $b)$ a special case is shown, where the variability associated with D4 only impacts a behavior chunk that is contained in same feature along with another behavior chunk, but with outgoing join relationships that have different decision items associated to them. Thus, in this case, whether the behavior that is composed by the join relationship associated with D4 becomes part of the derived product or not is actually completely independent of the JRs associated with D3 and D1. Therefore, the $cpath$ predicate of decision item D4 must only consider the outgoing JR of the impacted behavior chunk as part of its weaving path towards the commonality, which consequently yields the predicate $cpath(\text{D4}) = \text{D2}$.

Diagram $c)$ shows a case where a decision item impacts another decision item for which *partial realization* suffices (recall the *CC Dependency* attribute from Section 6.1.3; a more formal definition of these CC dependencies follows in this chapter). In particular, it impacts the part of a feature where the feature's decision item is only weakly associated. This means that $cpath(\text{D3}) = \text{D2}$ does not suffice as a $cpath$ predicate because the variable model elements

associated with D3 then would not become part of the derived product, when D2 is selected but D1 is deselected (which, in this case, is valid, as the JR that impacts D1 is only weakly associated). Therefore, the $cpath$ predicate—in this case—must go further and also include D1 as required, which eventually yields the predicate $cpath(\text{D3}) = \text{D2} \wedge \text{D1}$.

Diagram d) presents a case where decision item D5 is strongly associated with two join relationships that impact different features. The correct realization of decision item D5, thus, requires that it must become part of the commonality via both of these paths (as D5's CC dependency is *full realization*). This yields a $cpath$ predicate of $\text{D1} \wedge \text{D4}$ for D5, as the two impacted features both have only one dedicated decision item associated with their outgoing JRs. For a valid selection of decision item D4, however, *any realization* suffices. Note that the JR associated with D5 targets an abstract object within the aspect container of D4, while the outgoing JRs of this container have a scenario node as their source element. In this case, the abstract object will also be woven into both targets of D4's join relationships, as defined by ADORA's weaving semantics [Meier, 2009]. Hence, the shown scenario nodes and the object belong to the same model fragment. This differs from the case shown in diagram b), where two behavior chunks exist in one AC, which yield two different model fragments. As both targeted ACs of D4 again impact only one decision item, this yields a $cpath$ predicate of $\text{D2} \vee \text{D3}$ for D4. Transitively, the $cpath$ predicate of D5, thus, could also be interpreted as $\text{D1} \wedge ((\text{D4} \wedge \text{D2}) \vee (\text{D4} \wedge \text{D3}))$. Because every selection of a decision item must again satisfy this decision item's $cpath$ predicate (recall Equation 9.1) the simple formula $\text{D1} \wedge \text{D4}$ suffices and yields the same constraint.

Diagram e) shows a case where D5 impacts a feature associated with D4 that also includes a part of a feature that remains irrelevant for the $cpath$ of D5 (i.e., the nested AC that impacts D3). The reason is that this part of a feature can not further compose any model elements from its parent-AC (i.e., the AC where this AC is nested in), see [Meier, 2009]. Since decision item D4 only requires *any realization* (i.e., is weakly associated to all its JRs), the $cpath$ predicate of D5 must also further include the next target features, to ensure that the impact of D4 is eventually woven into a derived product. In theory, such cases can yield very large terms, when many parts of a feature with any realization are impacted. In the example shown, the $cpath$ predicate required for D5 is $(\text{D4} \wedge \text{D1}) \vee (\text{D4} \wedge \text{D2})$. Only $cpath(\text{D4})$ does not suffice as a $cpath$ predicate for D5 because that would mean that the functionality of D5 could also be applied with D1 and D2 set $false$. In fact, however, the variable model elements associated with D5 would not impact the derived product in any way, in such a case, because the only actual weaving paths of D5 lead via the decision items D1 and D2.

Diagram f) shows a case where D5 has multiple target-features and requires only *any realization*. Thus, the selection of either D3 or D4 already suffices for a valid selection of D5. Transitively, the selection of D4 requires only a *partial realization* in this case, which means that only the JRs to which D4 is strongly associated to are required (which is only one JR in this case) and the ones to which D4 is only weakly associated to are purely optional (i.e., they will be woven when the target model element is also selected and woven, but will not have any

impact on the selectability of D4, otherwise). Thus, D4's $cpath$ predicate must only include D1 because D2 yields no constraint in any form for D4.

Diagram g), finally, shows a case where decision item D7 impacts a feature that has two different decision items associated with its outgoing JRs. In this case D7 is strongly associated with only one JR and, thus, the $cpath$ predicate of D7 must only ensure that the target model element of this JR becomes part of the derived product at least once. The selection of either D5 or of D6, hence, already suffices, which leads to the predicate $cpath(\text{D7}) = \text{D5} \vee \text{D6}$. Transitively, D5 is weakly associated with all its JRs and requires only *any realization*. Therefore, if any of the JRs associated with D5 has a satisfiable $cpath$, D5 can validly be selected, which leads to the $cpath$ predicate $\text{D4} \vee \text{D3} \vee \text{D1}$ for D5. In this particular case, D4 is a dead decision item, however. This is the case because all JRs associated with D4 target an AC (or a model element within an AC) for which no weaving path exists through which it could eventually be composed into a derived product. Thus, $cpath(\text{D4})$ is always $false$ and this implies that D4 must always be $false$ for any satisfiable configuration. Transitively, this leads to an actual $cpath$ predicate of $\text{D3} \vee \text{D1}$ for D5 (as the path over D4 is a dead end). Further, D3 again requires only *any realization* and, thus, yields the predicate $cpath(\text{D3}) = \text{D2} \vee \text{D1}$. The $cpath$ predicates of D1 and D2 further are always $true$, as they directly impact the commonality.

Overall, the hierarchical dependencies of some of the decision items in these diagrams could already be argued as non-trivial to comprehend for human engineers. This makes an automated analysis support very recommendable. Even if an engineer prefers to manually reason about these dependencies, an automated analysis is beneficial, as a human engineer could easily miss a specific fact in his or her reasoning, which could possibly lead to an inaccurate conclusion. A tool would never miss any of these facts (i.e., if it is free of bugs) and, hence, always yields an accurate and precise analysis.

$cpath$ algorithms

The generation of all required $cpath$ predicates can be automated. An algorithm that generates all these $cpath$ predicates must consider all the special cases, as shown in Figure 9.3, to always generate a correct and precise $cpath$ predicate for any decision item. Internally, also the weaving paths of single join relationships need to be handled, in particular cases, as follows.

strong / weak DI associations

For generating correct $cpath$ predicates, the cross-cutting dependency types of how a decision item is strongly and/or weakly associated with its join relationships are essential—recall the *CC dependency* attribute as mentioned in Section 6.1.3. Depending on how the decision items are associated with their join relationship(s), the $cpath$ constraint must be stronger or weaker. Every decision item in SPREBA either falls into the category 'no cross-cutting' or can be categorized into one of these following three CC dependency types:

- *Full realization:* The decision item d is strongly associated to all its join relationships $J(d)$, which is the case when the following formula holds: $\bigwedge_{j \in J(d)} w(j) == false$. In this case, all targeted variable model elements must become part of the product, such that decision item d is fully realized (i.e., as required by these strong associations).

Algorithm 1 logical expression cpath(d)

/ This algorithm returns the $cpath$ expression for a specific decision item d, which is the required configuration of other decision items that must hold such that d can be set $true$ */*
$cpath$ = a logical expression */* the Boolean formula that will be returned */*
$J_{traversed}$ = {} */* the set of traversed JRs; avoids infinite loops in cyclic dependencies */*
T = {} */* the set of targeted model elements */*
if d requires *full realization* or *partial realization* */* as specified on page 149 */* **then**
  for every j in $J(d)$ where $w(j) == false$ **do**
    / every targeted model element $t(j)$ of every j where d is strongly associated ($\neg w(j)$) must eventually be composed into the commonality */*
    $T = T \cup t(j)$
    $J_{traversed} = J_{traversed} \cup j$
  end for
  / the $cpath$ predicate is the conjunction of required conditions for all targeted model elements in T */*
  $cpath = \bigwedge_{t \in T}$ conditionForRealization(t, $J_{traversed}$)
end if
if d requires *any realization* */* as specified on page 149 */* **then**
  for every j in $J(d)$ **do**
    / at least one targeted model element $t(j)$ must eventually be composed into the commonality */*
    $T = T \cup t(j)$
    $J_{traversed} = J_{traversed} \cup j$
  end for
  / the $cpath$ predicate is the disjunction of required conditions for all targeted model elements in T */*
  $cpath = \bigvee_{t \in T}$ conditionForRealization(t, $J_{traversed}$)
end if
return $cpath$

- *Partial realization:* The decision item d is strongly associated to some, but not all of its join relationships $J(d)$. Hence, it is strongly associated to at least one and also weakly associated to at least one of the JRs in $J(d)$. Partial realization is the case when the formula $\left(\bigvee_{j \in J(d)} w(j) == false\right) \wedge \left(\bigvee_{j \in J(d)} w(j) == true\right)$ holds. In this case all variable model elements that are targeted by join relationships to which d is strongly associated must become part of the product. All JRs to which d is only weakly associated do not need to be considered in the $cpath$ predicate.
- *Any realization:* The decision item d is weakly associated to all of its join relationships $J(d)$. Hence, there is no strong association of d to any of the JRs in $J(d)$. Any realization

is the case when the formula $\bigwedge_{j \in J(d)} w(j) == true$ holds for d. In this case, any (i.e., at least one, but no matter which one) of the targeted variable model elements must become part of the product, such that decision item d can be selected and actually also impacts the derived product.

Algorithm 2 logical expression conditionForRealization(t, $J_{traversed}$)

```
/* This algorithm returns the required configuration of other decision items that must hold
such that t becomes part of the commonality (anywhere and at least once) */
J_relevant = the set of all join relationships that compose t to anywhere else in the model
D_suffice = {} /* the set of decision items that suffice for realizing t */
recursiveCond = empty logical expression /* for the recursively composed condition */
for every j in J_relevant do
  if j ∉ J_traversed then
    /* every weaving path can traverse a join relationship only once; avoids loops */
    J_traversed = J_traversed ∪ j
    if w(j) == false then
      /* if j has a strongly associated decision item, then this decision item suffices */
      D_suffice = D_suffice ∪ d(j)
    else
      /* for weakly associated decision items it either can suffice, or another recursive step
      is necessary ... */
      if t(j) is part of the commonality then
        /* when t(j) is already part of the commonality, then d(j) (set true) suffices */
        D_suffice = D_suffice ∪ d(j)
      else if J(d(j)) ⊆ J_relevant then
        /* when the weakly associated decision item is only associated with JRs in J_relevant,
        then at least one of them must become part of the commonality when d is selected
        (any realization) and d suffices */
        D_suffice = D_suffice ∪ d(j)
      else
        /* not all J(d) are part of J_relevant, then d alone does only suffice when d together
        with the condition for one of the target elements of J(d) are satisfied */
        D_doesntSuffice = D_doesntSuffice ∪ d(j)
      end if
    end if
  end if
end for
```

(to be continued on the next page ...)

(continued from the previous page ... from **Algorithm 2**)
/ the selection of any decision item in $D_{suffice}$ or any path over a join relationship in $J_{candidates}$ suffices for t to become part of the commonality */*
for every d in $D_{doesntSuffice}$ **do**
  for every j in $J_{relevant}$ where $d(j) == d$ **do**
    if $recursiveCond$ == empty **then**
      $recursiveCond$ = ($d \wedge$ conditionForRealization($t(j)$, $J_{traversed}$))
    else
      $recursiveCond$ = $recursiveCond \vee (d \wedge$ conditionForRealization($t(j)$, $J_{traversed}$))
    end if
  end for
end for
if $recursiveCond$ == empty **then**
  return $\bigvee_{d \in D_{suffice}} d$
else
  return $\left(\bigvee_{d \in D_{suffice}} d\right) \vee recursiveCond$
end if

In general, the $cpath$ predicate is a formalization of the required configuration of other decision items that must hold, such that the decision item can validly be selected (i.e., set $true$). Figure 9.3 has presented several examples of how these $cpath$ predicates need to be defined in various special cases. In the following, we describe an algorithm that generates the required $cpath(d)$ predicate for a specific decision item in any arbitrary but valid SPREBA model.

$cpath(d)$ algorithm

Algorithm 1 shows how the complete hierarchical dependency of a specific decision item (i.e., the $cpath(d)$ predicate) can be computed automatically. The algorithm only requires the specification of the SPREBA model $\mathcal{M}$ and the decision item d which needs to be processed. The notation used is the formal notation as introduced in Section 9.1. Algorithm 1 is the main algorithm that computes the required target model elements that need to be woven into the commonality of a product, such that the decision item can be set $true$. Algorithm 2 further processes the conditions that must hold for such a required target model element, such that it eventually becomes part of the commonality. Since every decision item must satisfy its $cpath(d)$ predicate, whenever it is selected (recall Equation 9.1), the full $cpath$ predicate is defined transitively and the $cpath$ predicates for single decision items do not grow too large (recall the $cpath$(D7) predicate in diagram g) in Figure 9.3, for example). General descriptions and explanations are provided as comments in red color in the Algorithms 1 and 2. These algorithm descriptions, hence, should be well documented and reasonably understandable by themselves.

For Algorithm 2 the creation of the set $J_{relevant}$ is crucial. The details of how $J_{relevant}$ is computed had to be out of scope of this thesis, however. This calculation heavily depends on the used modeling language and weaving semantics. For ADORA, $J_{relevant}$ contains all JRs over

which the model element t can be composed to anywhere else. Not all outgoing JRs of the AC, where t is located within, also further compose t to anywhere else—recall the two separate behavior chunks in diagram b) in Figure 9.3, for example. Hence, $J_{relevant}$ sometimes only is a subset of all outgoing JRs of the targeted AC and must be derived for the particular language and composition semantics used (for ADORA, this can be straightforwardly derived from [Meier, 2009]).

Overall, the $cpath(d)$ predicate returned by Algorithm 1 is the complete hierarchy constraint for any decision item d. These $cpath$ predicates (i.e., for any decision item in a SPREBA model) are further used for generating the formula Φ, as specified in Equation 9.1. This formula Φ, along with any user-provided partial or full variability configuration, further is the key input for SPREBA's SAT-based automated constraints analysis, as outlined in Figure 9.1 and as follows.

9.4 Realizing SAT-Based Automated Constraints Analysis

The main activities supported by SPREBA's SAT-based automated analysis are (i) correctness verification of the product line domain model (i.e., with all decision items set $undecided$) and (ii) the calculation of constraint propagations for stepwise, incremental product derivation, as introduced in Chapter 8. The correctness verification of the domain model (Section 9.4.1) includes the maintenance of the model's satisfiability and the detection of dead and mandatory decision items. The automated analysis support for product derivation (Section 9.4.2) includes the calculation of constraint propagations, the calculation of minimal change sets when a user manually introduces a conflict and the undo of previous propagations that are not required anymore after a conflict was automatically resolved. Both use Boolean satisfiability (SAT) checks of the SPREBA model's equivalent Boolean formula $\Phi(\mathcal{M})_{withAssignm}$. Only for continuously assuring the domain model's satisfiability, as presented in Section 9.4.1, $\Phi(\mathcal{M})$ without a partial assignment suffices.
Other than these presented operations, SPREBA's tool implementation also supports further operations that are more straightforward, like counting the number of remaining satisfiable configurations for a specific decision setting (i.e., D_T and D_F), for example. Most operations as listed in Table 3.1 can also easily be realized with SPREBA models, once $\Phi(\mathcal{M})$ was generated.

9.4.1 Verifying the Domain Variability Model

maintaining satisfiability

To be able to perform any SAT-based automated analysis it is crucial that the SPREBA model and consequently the formula Φ is and stays *satisfiable* at all time. If the model itself is not satisfiable, no SAT-based automated analysis can be performed because every SAT check of Φ (with or without a partial assignment) will always return $false$. Hence, it is a primary objective

to maintain the logical truth of the domain model. While this may sound similar to Batory's idea of creating a logical truth maintenance system (LTMS) that is equivalent to a feature model, the idea of maintaining the domain variability model's satisfiability for any change has not explicitly been stressed in [Batory, 2005]. Batory's work primarily focuses on finding valid configurations, constraint propagation, and debugging feature models. He did not explicitly propose an approach that always assures the satisfiability of the variability model, as follows.

Maintaining the SPREBA Model's Satisfiability

To maintain a SPREBA model's satisfiability, every model edit operation (i.e., at modeling time) that changes the inherent variability model, needs to be checked for satisfiability. Theoretically, a change of any variability-relevant model element (recall Section 9.1) can change the inherent variability model. In practice, however, many changes in the model do actually not change the inherent variability model. All reasonable candidates for changes that have a high probability of changing the actual variability model are listed in Table 9.1. Whenever one of these change operations occurs in the model, then $\Phi(\mathcal{M})$ will be re-parsed to (i) verify whether the model has indeed changed and to (ii) re-verify the model's satisfiability SAT($\Phi(\mathcal{M})$) and re-calculate all automated analysis results, in case the model changed.
Note that the operations listed in Table 9.1 are not exhaustive. As shown in diagram $b)$ in Figure 9.3, it is possible that a behavior chunk has multiple outgoing JRs and that changing a state transition in such a behavior chunk splits the chunk into two separate behavior chunks. Such a change could exclude some of the JRs in $J_{relevant}$ for the $cpath(d)$ calculation of some decision items and, hence, also lead to a different formula $\Phi(\mathcal{M})$. Thus, these changes also need to be considered as changes of the variability model. As such changes heavily rely on the actually used modeling language and composition semantics, this is considered out of scope. For ADORA, these cases can straightforwardly be derived from [Meier, 2009]. Or, when a exhaustive tool implementation exists, this impact can also directly be verified in the tool (i.e., by composing the relevant join relationships and detecting whether the element of interest t was actually included in the composition, or not).
Other model edit operations that only change the requirements model, but do not change any variability-relevant model elements—except for cases as mentioned above—, are not supposed to change the model's equivalent formula $\Phi(\mathcal{M})$. Hence, they also do not require a repetition of the hitherto automated analysis (i.e., the pre-calculated constraint propagations shown in the columns *ifTrue*, *ifFalse*, etc., in the decision table view, recall Section 6.1.3 and Chapter 8).

variability-relevant edit operations

Table 9.1 presents a complete listing of model changes in an ADORA and SPREBA model that can potentially change the variability model. These edit operations include any changes of decision items, of variable join relationships, and of C and VP constraints. Changes of aspect containers do not directly change the variability model. For example, changing whether an AC is variable or not (i.e., changing its variability attribute $b(a)$ between $true$ and $false$), or

Table 9.1: A listing of model edit operations of variability-relevant model elements that change a model's actual variability that constitutes $\Phi(\mathcal{M})$.

Changed Element	**Description**	**Change (formally)**
decision item d	adding a new decision item	$D' = D \cup \{d\}$
	removing a decision item	$D' = D \setminus \{d\}$
	adding or removing decision item associations	$J'(d) \neq J(d)$
	changing between strong and weak association of a decision item with a JR	$\bigvee_{d \in D} \bigvee_{j \in J(d)} w'(j) \neq w(j)$
variable join relationship j	adding a new outgoing JR to a variable feature	$J' = J \cup j \mid \bigvee_{a \in A} b(a) == true \wedge (s(j) \in C(a))$
	re-routing an outgoing JR from or to a different model element[1]	$\bigvee_{j \in J} (s'(j) \neq s(j)) \vee (t'(j) \neq t(j))$
	deleting a variable JR	$J' = J \setminus \{j \mid d(j) \neq \emptyset\}$
variation point constraint v	adding a new VP constraint	$V' = V \cup \{v\}$
	changing the specification of an existing VP constraint	$\bigvee_{v \in V} (D'(v) \neq D(v)) \vee (n'(v) \neq n(v)) \vee (m'(v) \neq m(v))$
	deleting a VP constraint	$V' = V \setminus \{v\}$
Boolean algebra constraint c	adding a new C constraint	$C' = C \cup \{c\}$
	changing the specification of an existing C constraint	$\bigvee_{c \in C} (t'_a(c) \neq t_a(c)) \vee (o'(c) \neq o(c)) \vee (t'_c(c) \neq t_c(c))$
	deleting a C constraint	$C' = C \setminus \{c\}$

deleting a variable AC as a whole, does not directly impact the variability model, but *does* trigger an automated addition of a new decision item, a removal of decision item associations to outgoing join relationships, or a deletion of one or more variable join relationships, as a consequence. These consequences are again covered in Table 9.1 and will also trigger a re-parsing of the model into $\Phi(\mathcal{M})$, hence. Note that not every operation as listed in Table 9.1 must necessarily yield a different variability model. A re-parsing of Φ must not be done for any change, but only for those changes that change $\Phi(\mathcal{M})$. Any other change, hence, still maintains the model's satisfiability.

[1]Re-routing a JR to a different model element within the same aspect container might suffice to alter the variability model, recall, e.g., diagram b) in Figure 9.3.

fine-tuning Table 9.1

In general, the generation process of $\Phi(\mathcal{M})$ out of an existing SPREBA model can be considered a model operation. Thus, the variability-relevant model elements that are 'touched' by such a model operation in a SPREBA model can also be considered as those that are part of the *footprint* of this operation, as described in [Jeanneret et al., 2011]. The detailedness of the edit operations listed in Table 9.1 (for ADORA and SPREBA) is not absolutely precise, but a very precise version of the table could be generated by using the ideas from Jeanneret et al.'s footprinting concept. It could be, for example, that static footprints already yield very precise data. This could then be refined with dynamic footprinting for special cases, if still needed.

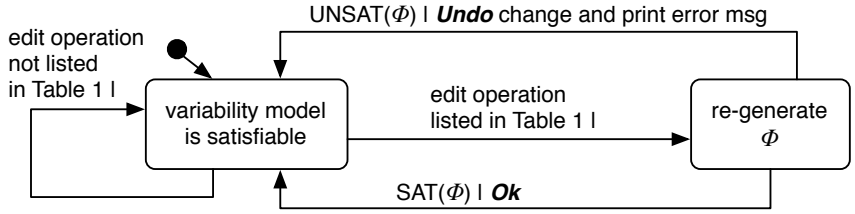


Figure 9.4: Automated evaluation of every relevant model edit operation (Table 9.1) to guarantee the variability model's satisfiability.

rejecting unsatisfiable changes

Figure 9.4 shows how SPREBA only allows models that are correct by construction. Every time the SPREBA domain model $\mathcal{M}$ is changed, the change is classified on whether it yields a different formula $\Phi(\mathcal{M})$. When the variability model actually has changed (i.e., the *edit operation* is *listed in Table 9.1*, as shown in Figure 9.4), then $\Phi(\mathcal{M})$ is re-generated for the new variability model and checked for satisfiability. In this state, while Φ is re-generated, the satisfiability of the variability model is temporarily in an unknown state, until $\Phi(\mathcal{M})$ is created and its satisfiability SAT($\Phi(\mathcal{M})$) was evaluated. When satisfiable, then the change is valid and the domain engineering process can continue. When unsatisfiable (i.e., UNSAT(Φ) or $\neg$SAT(Φ)), then the recent edit operation has introduced a conflict in the variability model's constraints, that lead to a model where no configuration exists that satisfies all constraints (i.e., $\Phi(\mathcal{M})$). A simple example for such an edit operation that causes a conflict would be adding the C constraint D1 $\Leftrightarrow \neg$D1. This change would be verified as unsatisfiable and would be undone immediately, as shown in Figure 9.4. A customized explanation would be presented to the user, which clarifies why the most recent change has caused unsatisfiability and could not be maintained. The SPREBA approach, hence, immediately detects whether a change causes unsatisfiability, only allows those changes that maintain satisfiability and, thus, can guarantee that every created variability model is *satisfiable*.

Validity of Variability Models

dead or mandatory DIs

A variability model that contains dead decision items (i.e., that always lead to unsatisfiability when set $true$) or mandatory decision items (i.e., that always lead to unsatisfiability when bound to $false$) is not a fully valid model. Such variability models as a whole are satisfiable, however. Product line engineers should be aware of any dead or mandatory features that are in a domain model and should be urged to either delete them (for dead decision items), to compose and remove them (for mandatory decision items), or to refactor the variability model and its constraints to make this decision item variable again (such that both a selection and a deselection still allows a remaining satisfiable full configuration). Leaving dead or mandatory decision items (and their respective features and parts of a feature) in the model, although they are in fact not variable, makes the variability model less effective and leads to a not fully valid variability model, hence. Luckily, SPREBA's SAT-based automated analysis makes it really easy to automatically identify such dead or mandatory decision items.

Algorithm 3 D deadDecisionItems($\Phi(\mathcal{M})$, $\mathcal{M}$)

```
/* This algorithm returns the set of dead decision items D_dead (i.e., those that cannot be set
true without causing unsatisfiability) in any SPREBA model M. */
D_dead = ∅
for every d in D(M) do
  if SAT(Φ(M) ∧ d) = false then
    /* The variability model is not satisfiable with d bound to true. Hence, d is 'dead'. */
    D_dead = D_dead ∪ d
  end if
end for
return D_dead
```

Building on the Boolean formula $\Phi(\mathcal{M})$ and using an off-the-shelf SAT solver makes the detection of dead and mandatory decision items quite straightforward, as shown in the Algorithms 3 and 4. These algorithms essentially add a single decision item as selected (in Algorithm 3) or deselected (in Algorithm 4) as a partial assignment to the formula Φ and run a SAT check. Unsatisfiability in this context is already a proof for a dead decision item (i.e., when selecting this decision item with $true$ is not satisfiable) or a mandatory decision item (i.e., when deselecting it with $false$ is not satisfiable) has been found. The Algorithms 3 and 4 also include further textual documentation in red color.

Algorithm 4 D mandatoryDecisionItems($\Phi(\mathcal{M})$, $\mathcal{M}$)

/ This algorithm returns the set of mandatory decision items $D_{mandatory}$ (i.e., those that cannot be set $false$ without causing unsatisfiability) in any SPREBA model $\mathcal{M}$. */*
$D_{mandatory}$ = $\emptyset$
for every d in $D(\mathcal{M})$ **do**
  if $SAT(\Phi(\mathcal{M}) \wedge \neg d) = false$ **then**
    / The var. model is not satisfiable with d bound to $false$. Hence, d is 'mandatory'. */*
    $D_{mandatory}$ = $D_{mandatory} \cup d$
  end if
end for
return $D_{mandatory}$

9.4.2 Constraint Propagation

A satisfiable variability model, which a SPREBA model always is, allows the straightforward calculation of required constraint propagations for a specific variability binding decision, based on an off-the-shelf SAT solver. The automated calculation of constraint propagations (also called constraint propagation previews in the decision table's columns *ifTrue*, *ifFalse*, and *ifUndecided*, recall Section 6.1.3) is crucial when deriving a product from a SPREBA product line model. Constraint propagation is particularly relevant during a stepwise, incremental product derivation, recall Chapter 8, or to realize a staged configuration (i.e., a special case of stepwise, incremental product derivation), recall Section 3.3.1.

resolving conflicts

The algorithmic solution for calculating the set of required propagations, when binding an unbound decision item ($undecided \rightarrow true|false$), is rather straightforward. The here presented solution, however, also computes the required constraint propagations for *any* variability binding decision ($undecided|true|false \rightarrow undecided|true|false$) at any time during a derivation (i.e., for any partial assignment of decision items—D_T and D_F). This means that any decision that was already propagated as a consequence of a previous decision (i.e., that was automatically set, recall Section 6.1.3) can still be changed manually by a user, which by definition causes unsatisfiability. In such a case the presented algorithm first detects a minimal set of required changes in the previously manually set decisions, which restores the satisfiability of the configuration. Then, it re-calculates the still required and additionally required constraint propagations because of these changed decisions. This completely restores both the satisfiability and the validity of the currently set configuration, for any change that introduced a conflicting configuration. Thus, *any* manual change in a configuration will always again lead to a fully satisfiable model. In every reachable variability configuration all required constraint propagations will always be set as well (as already illustrated in [Batory, 2005]). Any previously propagated decisions that are not required as propagations anymore, because a previous man-

ual decision was reversed during the automated resolution of a conflict, will also be reversed back to $undecided$. Compared to our already published material, as presented in [Stoiber and Glinz, 2010b], for example, this automated resolution of configuration conflicts is novel. In state-of-the-art work the automated resolution of conflicts during variability configuration was to date still treated as an unsolved and open problem, see, e.g., [Nöhrer and Egyed, 2010]. Hitherto major references on SAT-based automated variability analysis, see [Janota, 2010] or [Benavides et al., 2010], for example, have also left this problem unsolved.

Algorithm 5 presents the main algorithm to calculate constraint propagations. Detailed explanations are provided in the algorithm description in red color. Whenever a new decision is taken in the decision table, Algorithm 5 includes this decision into the current decision setting (D_T, D_F) and evaluates for every other decision d_{test} whether d_{test} can still be set $true$ and $false$ and still yields a satisfiable configuration. If both settings of d_{test} are satisfiable, then no constraint propagation is necessary. If $d_{test} = true$ is not satisfiable, but $d_{test} = false$ is, then d_{test} must be propagated to $false$ because then the SPREBA model's constraints (encoded in Φ) and its current partial assignment (D_T and D_F) are only compatible with $d_{test} = false$. If $d_{test} = false$ is not satisfiable, but $d_{test} = true$ is, then d_{test} must be propagated to $true$, respectively. The set of required constraint propagations is recorded and returned as $P_{d,v(d)}$, which contains all the decision item and decision value pairs that constitute the constraint propagation that is required for setting $d_{toEvaluate}$ to $v_{toEvaluate}$. If both $false$ and $true$ are unsatisfiable for a decision item d_{test}, then setting $d_{toEvaluate}$ to $v_{toEvaluate}$ has introduced a conflict with the already set partial assignment (i.e., the previously taken decisions $D_{manuallySet}$) and must be resolved with a recursive algorithm, by finding a minimal set of changes in these previously manually taken decisions $D_{manuallySet}$, among which the conflict must be, if it is not a dead or mandatory decision item that the user just intended to change. Algorithm 5 shows the general procedure for constraint propagation (i.e., for the decisions $undecided \rightarrow true|false$). The automated resolution of conflicting decisions (i.e., from $true|false \rightarrow true|false|undecided$, where $v(d)' \neq v(d)$) is further handled in Algorithm 6.

finding a minimal satisfiable change set

Algorithm 6 illustrates how conflicts are resolved in interactive configuration of a SPREBA model. Essentially, a conflict can always be traced back onto the manually set decision items, since automatically set decisions (i.e., decisions that were set as previous propagations of previous user-taken decisions) are merely a propagation of the consequences of such manually taken decisions. When a propagation of a previously manually set decision item is changed, this implies a conflict (i.e., an unsatisfiable configuration, as verified previously, when the decision was propagated) or an inconsistent decision setting, in case the propagated decision was reversed to $undecided$ (i.e., this configuration is still satisfiable, but not fully valid, because the propagated decision can only take the value as previously propagated, for a satisfiable product). However, the conflict can also be with multiple already manually taken decision items.
Algorithm 6 starts to exclude single decision items from the set of previously manually taken decision items $D_{manuallySet}$ and checks for satisfiability. If the remaining manually set decision

Algorithm 5 $P_{d,v(d)}$ calcConstrProp($\Phi(\mathcal{M})$, $\mathcal{M}$, D_T, D_F, $d_{toEvaluate}$, $v_{toEvaluate}$)

Require: $\Phi(\mathcal{M})$ is the variability model in Boolean logic, $\mathcal{M}$ is the model as a whole, D_T is the set of all decision items currently set $true$, D_F is set of all set $false$, respectively, $d_{toEvaluate}$ is the decision item to calculate the constraint propagation for, and $v_{toEvaluate}$ is the truth value $\{true|false|undecided\}$ for which the constraint propagation is calculated.

/ An algorithm that returns all required constraint propagations $P_{d,v(d)}$ (i.e. the further required decisions) such that decision item $d_{toEvaluate}$ can be set to $v_{toEvaluate}$, the model is satisfiable, and all the remaining $undecided$ decision items are satisfiable in both cases, when set $true$ and when set $false$. */*

$P_{d,v(d)} = \emptyset$ */* the set of required constraint propagations to return */*

$v_{toEvaluate} \Rightarrow (D_T = D_T \cup d_{toEvaluate})$ */* add $d_{toEvaluate}$ to D_T when set $true$ */*

$\neg v_{toEvaluate} \Rightarrow (D_F = D_F \cup d_{toEvaluate})$ */* ... or to D_F when set $false$ */*

$(v_{toEvaluate} = undecided) \Rightarrow (d_{toEvaluate} \notin \{D_T, D_F\})$ */* ... or to none of them */*

if $v_{toEvaluate} \neq undecided$ **then**

  for every $d_{test} \in \{D(\mathcal{M}) \setminus \{D_T \cup D_F \cup d_{toEvaluate}\}\}$ **do**

    / Evaluate for the current configuration (including $d_{toEvaluate}$ set to $v_{toEvaluate}$) whether setting an $undecided$ DI d_{test} set $true$ (i.e., b_t) or $false$ (i.e., b_f) is still satisfiable */*

    $b_t = \text{SAT}(\Phi_{withAssignm}(\mathcal{M}, D'_T, D_F) \mid D'_T := D_T \cup d_{test})$

    $b_f = \text{SAT}(\Phi_{withAssignm}(\mathcal{M}, D_T, D'_F) \mid D'_F := D_F \cup d_{test})$

    / Depending on how b_t and b_f evaluate: record a required constraint propagation, calculate a minimal change set to resolve the conflict (if both are unsatisfiable) or do nothing, if both are satisfiable */*

    if $(b_t \wedge \neg b_f)$ **then**

      $P_{d,v(d)} = P_{d,v(d)} \cup (d_{test}, true)$

    else if $(\neg b_t \wedge b_f)$ **then**

      $P_{d,v(d)} = P_{d,v(d)} \cup (d_{test}, false)$

    else if $(\neg b_t \wedge \neg b_f)$ **then**

      / A previously propagated decision was changed and caused a conflict: a minimal set of changes of previously set DIs needs to be found and post-processed */*

      return $incExhaustiveSearch(\Phi(\mathcal{M}), \mathcal{M}, D_T, D_F, d_{toEvaluate}, v_{toEvaluate})$

    end if

  end for

else

  return $incExhaustiveSearch(\Phi(\mathcal{M}), \mathcal{M}, D_T, D_F, d_{toEvaluate}, v_{toEvaluate})$

end if

return $P_{d,v(d)}$

items plus the newly set decision $d_{toEvaluate}$ are again satisfiable, then this set of decision items is a minimal change set among the manually set decision items. If no such minimal change set

Algorithm 6 $P_{d,v(d)}$ incExhaustiveSearch(($\Phi(\mathcal{M})$, $\mathcal{M}$, D_T, D_F, $d_{toEvaluate}$, $v_{toEvaluate}$))

Require: /* Same as in Algorithm 5 */

/ The decision item settings of $D_{manuallySet}$ and $d_{toEvaluate}$ together are unsatisfiable. This algorithm searches for a minimal set of reversions of decisions in $D_{manuallySet}$ back to $undecided$ such that $d_{toEvaluate} \cup D_{manuallySet}$ again yield a satisfiable partial assignment. Afterwards, it performs a post-processing to remove unnecessary previous propagations in D_T and D_F (as manually set decisions were reversed to $undecided$) and adds all newly required propagations for the new partial assignment. Also the generalizations required when $d_{toEvaluate}$ is reversed to $d_{toEvaluate} = undecided$ are calculated. */*

$D_{test} = \emptyset$, $P_{d,v(d)} = \emptyset$, $D'_T = \emptyset$, $D'_F = \emptyset$

$D_{manuallySet}$ = the set of all user-set decision items */* ...all decisions not set $undecided$ and not automatically propagated (i.e., not* automatically set*, recall Section 6.1.3) */*

D_{fixed} = the set of fixed user-decisions */* ...these decisions must not change when a conflict is resolved automatically (recall the* Fixed *column in Section 6.1.3) */*

/ searching a minimal satisfiable set of decision undos in $D_{manuallySet}$ */*

if $v_{toEvaluate} = undecided$ **then**

/ if the decision $d_{toEvaluate}$ was reversed to $undecided$ and is not satisfiable when set $true$ and $false$ (which is required), then this conflict has to be resolved */*

$\neg\text{SAT}(\Phi_{withAssignm}(\mathcal{M}, D'_T, D_F) \mid D'_T := D_T \cup d_{toEvaluate}) \Rightarrow (v_{toEvaluate} = true)$

$\neg\text{SAT}(\Phi_{withAssignm}(\mathcal{M}, D_T, D'_F) \mid D'_F := D_F \cup d_{toEvaluate}) \Rightarrow (v_{toEvaluate} = false)$

end if

if $v_{toEvaluate} \neq undecided$ **then**

for ($i = 1 \ldots |D_{manuallySet}|$ until $P_{d,v(d)} \neq \emptyset$) **do**

/ for any i-tuple of decision items in $D_{manuallySet}$ */*

for (every $D_{test} \in_{randomSelected} \binom{D_{manuallySet}}{i}$ until $P_{d,v(d)} \neq \emptyset$) **do**

/ test whether the conflicting decision(s) between ($D_{manuallySet}$ and $d_{toEvaluate}$ are those in D_{test} (without considering the previous propagations in D_T and D_F) */*

if SAT($\Phi_{withAssignm}(\mathcal{M}), (D_{manuallySet} \backslash (D_{test} \cup D_{fixed} \cup d_{toEvaluate}))$) **then**

/ if yes: then undoing the dec.s in D_{test} resolves the conflict with $d_{toEvaluate}$ */*

$\forall_{d \in D_{test}} P_{d,v(d)} = P_{d,v(d)} \cup \{d, undecided\}$ */* restore satisfiability */*

$D_{manuallySet} = D_{manuallySet} \setminus D_{test}$ */* they are not manually set anymore */*

for every $d \in D_{manuallySet}$ **do**

$v(d) \Rightarrow (D'_T = D'_T \cup d) \wedge \neg v(d) \Rightarrow (D'_F = D'_F \cup d)$

end for

$v_{toEvaluate} \Rightarrow (D'_T = D'_T \cup d_{toEvaluate}) \wedge \neg v_{toEvaluate} \Rightarrow (D'_F = D'_F \cup d_{toEvaluate})$

end if

end for

end for

end if

(to be continued on the next page ...)

(continued from the previous page ... from **Algorithm 6**)
/ Post-processing: the new $D_{manuallySet}$ (without the reversed decisions D_{test}) together with $d_{toEvaluate}$ may require new propagations and may make previous propagations obsolete */*
if $(P_{d,v(d)} \neq \emptyset) \mid v_{toEvaluate} == undecided)$ **then**
  if $v_{toEvaluate} \neq undecided$ **then**
    for every $d \in (D(\mathcal{M}) \backslash (D_{manuallySet} \cup D_{fixed} \cup d_{toEvaluate}))$ **do**
      / if the required propagation not already set this way, add to propagation */*
      if $v(d) \neq false \wedge \neg\text{SAT}(\Phi_{withAssignm}(\mathcal{M}, (D'_T \cup d), D'_F))$ **then**
        $(d \notin D_F) \Rightarrow (P_{d,v(d)} = P_{d,v(d)} \cup \{d, false\})$
      else if $v(d) \neq true \wedge \neg\text{SAT}(\Phi_{withAssignm}(\mathcal{M}, D'_T, (D'_F \cup d)))$ **then**
        $(d \notin D_T) \Rightarrow (P_{d,v(d)} = P_{d,v(d)} \cup \{d, true\})$
      end if
    end for
  end if
  for every $d \in ((D_T \cup D_F) \backslash (D_{manuallySet} \cup D_{fixed} \cup d_{toEvaluate})) \mid d = \{true, false\}$ **do**
    / if the hitherto propagation is not required anymore, propagate back to undec. */*
    $v_{toEvaluate} = undecided \Rightarrow D'_T := D_T \wedge D'_F := D_F$
    if $\text{SAT}(\Phi_{withAssignm}(\mathcal{M}, (D'_T \cup d), D'_F)) \wedge \text{SAT}(\Phi_{withAssignm}(\mathcal{M}, D'_T, (D'_F \cup d)))$ **then**
      $P_{d,v(d)} = P_{d,v(d)} \cup \{d, undecided\})$
    end if
  end for
end if
return $P_{d,v(d)}$

is found for one change only, then all combinations of two previously manually set decision items are evaluated, and so forth. Only those decision items whose selection was fixed in the decision table (i.e., which are part of D_{fixed}, recall the *Fixed* column as introduced in Section 6.1.3) are not evaluated, but keep their fixed value. It is possible, however, that a set of taken and fixed decisions leads to unsatisfiability in any case (i.e., that no satisfiable change set can be found to restore an overall satisfiability). In such a case an error message is presented to the user, which prompts her to unfix some of these fixed decision items.
When no decision items are fixed and no dead or mandatory decision items exist, then a minimal satisfiable change set will always be found for any possible change. However, there can also be multiple minimal change sets for resolving a conflict. In such a case our solution randomly chooses one of them. A real random-selection of one minimal change set may actually be advantageous because the tool does not start to systematically toggle between particular satisfiable subsets of configurations, but still keeps a wider set of possible satisfiable configurations reachable. In general, as long as the reduced set of $D_{manuallySet} \cup d_{toEvaluate}$ (with D_{test} removed as a candidate minimal change set) remains unsatisfiable, a larger set of changes is generated and tested, until eventually a satisfiable change is found.

The only two cases where no satisfiable change is possible, as mentioned, are (i) when the decision item was already a dead or mandatory decision item in the SPREBA domain model and (ii) when all other decision items in conflict are fixed to their decision value. In the first case, the decision field for such decision items, thus, must be deactivated during a product derivation, as this decision value is not allowed to be changed anyways. In the second case, no constraint propagation is recorded and the user will be prompted that this decision can not be taken because of other already taken and fixed decisions (i.e., whenever a propagated decision is fixed by a user it will also be considered a manually set one).
In the case where $v_{toEvaluate}$ was reversed back to $undecided$, this does not necessarily cause a conflict, but requires that $d_{toEvaluate}$ can be set $true$ and $false$ and that a satisfiable full configuration must exist for both cases. Finally, after the minimal change set was found and recorded as a constraint propagation, a post-processing finalizes the adaption of the previously set constraint propagations to this new setting.

post-processing a conflict resolution

The post-processing, as presented in Algorithm 6, already starts from a satisfiable configuration and, hence, performs two tasks: (i) it records and adds all newly required propagations for the new decision setting and (ii) it undoes all hitherto set constraint propagations that are not necessary anymore because the decision(s) that originally caused them may not be set this way anymore. Therefore, it first checks for all remaining decision items (except for $d_{toEvaluate}$ and those in $D_{manuallySet}$ and D_{fixed}) whether they need to be propagated to either $true$ or $false$, if they are not already set this way. Then, it also verifies whether all bound (i.e., previously propagated) decision items (again except for $d_{toEvaluate}$ and those in $D_{manuallySet}$ and D_{fixed}) are still required as propagated. If not, they will safely be reverted back to $undecided$ and will also be added to the set of constraint propagations $P_{d,v(d)}$.

related work

Related work, e.g., [Trinidad et al., 2008], [White et al., 2008], or [White et al., 2009], did neither address nor solve the problems addressed in this chapter, as already discussed in Section 3.3.1. Only recently, [Xiong et al., 2011] has presented an approach that is similar to what was presented in this section.

conclusion

This solution, as presented in the Algorithms 5 and 6, allows processing *any* change of *any* variability binding decision at *any* time and always yields an again satisfiable configuration that keeps as many of the previously manually taken decisions as possible. Further, the presented solution is complete, but still takes a brute force approach—to some extent—to find the minimal satisfiable change sets. This slows the runtime performance, possibly unnecessarily. These parts of the presented algorithms may still need to be improved in future research, therefore, as also highlighted in Section 15.3. Such improvements could use the calculation of minimal unsatisfiable cores, for example, which modern SAT solvers can generate. An in-depth performance evaluation this SAT-based analysis solution for SPREBA follows in Chapter 13.

CHAPTER 10

Feature Unweaving: Efficiently Creating Product Line Models

The design of the variability of a software product line is crucial to its success and evolution. Meaningful variable features need to be elicited, analyzed, documented, and validated, when an existing software system evolves into a software product line. This requires the creation of the variability model and a continuous refinement and maintenance of the defined variable features, while the product line is developed and evolved.

related work

Recent work has introduced tool support for improving the evolution of an existing portfolio of products into a software product line by mining the requirements specifications of existing valid product configurations and automatically creating a feature model that represents the common and variable features [Wang et al., 2009] [Weston et al., 2009]. If multiple consistent product specifications do not yet exist, however, such a method can not be applied.
Other work has addressed the problem of reverse engineering a software product line from product models. For example, Rubin and Chechik did so by employing a heterogeneous match and refactoring algorithm, called ThreeVaMar, for UML models [Rubin and Chechik, 2010]. Zhang et al. also showed how CVL models can be created by clone detections in existing product models [Zhang et al., 2011]. Their product models can be written in any modeling language that has been defined based on the Meta Object Facility (MOF). Zhang's work was motivated by the observation that companies often synthesize their product lines from a series of existing products and that differences in product models often directly reflect the variability of the domain in the problem space. Further, She et al. presented a solution for reverse engineering

feature models from dependencies and textual feature descriptions [She et al., 2011]. Their solution does not necessarily require the descriptions of concrete products. All of these existing approaches, however, build on modeling languages that use a scattered concrete syntax (e.g., the UML) and orthogonal variability modeling. SPREBA, contrarily, relies on an integrated variability modeling approach. SPREBA's feature unweaving primarily aims at supporting domain engineers during the otherwise fully manual variability model design and evolution tasks, rather than automating the process as a whole.

semi-automated variability extraction

In the ideal case a software product line is systematically planned and developed as soon as its market opportunity or demand emerges. Building a software product line with a smaller scope initially and evolving it incrementally has been found less expensive than extracting a product line from already existing products or building one with its full scope from scratch [Krueger, 2002]. As soon as a second closely related product gets developed, commonality and variability emerge and the existing requirements model can be regarded a *reference model* for the product line. Manually creating an aspect-oriented software product line model from a reference model requires considerable clerical and intellectual efforts, however. It requires numerous manual model edit operations for every variable feature and a thorough understanding of aspect-oriented modeling (i.e., or the particular compositional approach that is used). Additionally, manual variability modeling is also prone to errors and mistakes.

This chapter briefly introduces *feature unweaving*, a novel approach that allows a semi-automated variability extraction of variable features in an ADORA model. Feature unweaving was originally developed by [Jehle, 2010], demonstratively presented in [Stoiber et al., 2010] and comprehensively presented in [Stoiber and Glinz, 2010a]. Feature unweaving considerably eases the required efforts for variability model creation with SPREBA.

10.1 Support for Product Line Model Creation

realizing feature unweaving

In general, feature unweaving supports domain requirements engineers in incrementally evolving the given product or reference model into a software product line model. It allows a product line requirements engineer to efficiently and semi-automatically evolve a reference model (that satisfies the prerequisites listed in Chapter 5) into a SPREBA product line model: when he or she has identified and selected the variable model elements that constitute a variable feature, feature unweaving automatically extracts these elements and refactors them into a feature, using aspect-oriented modeling. While the domain requirements engineer only needs to provide this selection of model elements, the feature unweaving function fully automatically performs the actual extraction and aspect-oriented variability specification. This allows a new, semi-automated and incremental style of identifying and specifying variable features.
Feature unweaving performs four steps fully automatically: (i) it creates a feature aspect and builds all the necessary internal aspect structure, (ii) it removes all selected elements from the

requirements model, (iii) it inserts them into their respective aspect, and (iv) it builds all the necessary weaving semantics. The resulting refactored model is semantically equivalent to the original model. This equivalence can easily be verified by executing a weaving operation of the newly unwoven aspect and comparing the resulting model with the original one. Mistaken or not ideal feature extractions can also be undone, by simply weaving and removing them again.

benefits

Feature unweaving significantly reduces the required effort for variability specification, both on a clerical and intellectual level. Using feature unweaving to semi-automatically extract variable features also guarantees the absence of additional syntactic and semantic mistakes, which guarantees that the resulting model is *correct by construction*. This correctness-by-construction can easily be verified by re-weaving an extracted feature and comparing it to the original model. It could also be assured by fully verifying the feature unweaving implementation, which would then always assure either a correct extraction or a correctly identified not extractable selection. The latter occurs when a provided selection of model elements can not be formulated with aspect-oriented modeling and would, thus, violate the algorithm's pre-conditions. Such not extractable selections, however, were found to be rather rare in ADORA [Jehle, 2010].
Variable features also often affect several modules and facets of a system in heterogeneous and sometimes also homogeneous ways (explanations of these terms follow). ADORA aspects also allow an explicit specification of such heterogeneously scattered concerns and are particularly well suited for homogeneous cross-cutting concerns, which are the classic *aspects* (e.g., like logging or authentication). When selected model elements are extracted from other variable features, the feature hierarchy constraints (as specified in Section 9.3) will also be created automatically. By default the strongest possible constraint gets chosen (strong dependencies for all JRs), which then gets relaxed by the user to weak dependencies, wherever feasible. We chose to generate strong dependencies by default because we found that product line models typically are constrained too little, rather than too much (i.e., billions of products are often technically allowed for large variability models, while only much fewer may actually make sense and be feasible for most real-world markets).
Our hitherto empirical results showed that feature unweaving yields significant improvements in the efficiency of product line specification, product variant definition, release planning, and consistency maintenance activities, when compared to an ad-hoc approach with AND/OR tables, as presented in [Stoiber and Glinz, 2010a]. Other empirical results also provided evidence of a significant reduction of the complexity and necessary size of the graphical product line requirements specification [Zoller, 2010].

types of variable features

In previous research we have identified three different types of variable features in an integrated model [Stoiber and Glinz, 2010a]. We distinguish between local, homogeneously cross-cutting, and heterogeneously cross-cutting variable features. This terminology is older, however. In [Colyer and Clement, 2004] and in [Colyer et al., 2004], for example, homogeneous and heterogeneous cross-cutting concerns were already explicitly addressed. [Apel et al., 2006] introduced such a distinction as well. Our definitions are as follows:

1. *local features:* the variable model elements that constitute this feature are located in only one component in the commonality or in an existing variable feature;
2. *homogeneously cross-cutting features:* the variable model elements occur in exactly the same form in multiple different components of the commonality and/or in different variable features (these are similar to classic cross-cutting concerns as addressed in [Meier, 2009]);
3. *heterogeneously cross-cutting features:* the variable model elements occur in a different form in multiple different components of the commonality and/or in different variable features.

The feature unweaving approach must be capable of semi-automatically extracting all of these. The vast majority of variable features are local and heterogeneously cross-cutting ones, as also hitherto empirical results by Zoller showed [Zoller, 2010]. Extracting such variable features (i.e., such selections of model elements) from a reference model is the task of the feature unweaving solution. Homogeneously cross-cutting features are quite rare, as we found in our hitherto experience (Chapters 12 and 14). The current realization of feature unweaving does not particularly support the direct extraction of homogeneously cross-cutting features, but rather handles them like heterogeneously cross-cutting ones, which is semantically correct, but leads to redundancy. Support for extracting homogeneously cross-cutting features could be realized by either (i) merging all redundant model elements after such an extraction, or by (ii) requiring only the selection of one instance, an automatic search and removal of all further equivalent instances, and the creation of adequate join relationships to all locations where this cross-cutting concern occurred (which would be the more practical solution). Because homogeneously cross-cutting variable features are very rare a comprehensive implementation for semi-automatically extracting them has not yet been developed in the ADORA tool and is still subject to future work.

10.2 Algorithms and Illustration

feature unweaving algorithm

Figure 10.1 shows the structure of the main feature unweaving algorithm and its recursive extraction function, as presented in [Stoiber and Glinz, 2010a]. This description shows only the abstract behavior of this function, but does not illustrate how exactly the weaving semantics is created. Jehle has originally realized an iterative implementation of the feature unweaving function and specified its behavior with activity diagrams [Jehle, 2010]. Jehle's conceptual solution has focused on realizing feature unweaving for ADORA. The solution presented in [Stoiber and Glinz, 2010a] and in Figure 10.1 generalizes [Jehle, 2010]'s descriptions and is a recursive description. It omits the specifics of the language and composition semantics used, for brevity. This recursive description is more succinct and reflects the nature of this extraction problem more naturally.

Main Feature Unweaving Algorithm:

1. *Create the feature container*
2. *Call the recursive extraction function (parameters: model, feature container)*
3. *Remove all redundant ACs in the feature*
4. *Fine-tune the graphic layout*
5. *Declare the feature container as a variable (this adds and associates a new unique decision item to all outgoing join relationships)*

Recursive Extraction Function:

Input-parameters: a PC, an AC
2.1 *Find the basePC*
2.2 *Extract all directly nested selected elements of the basePC into this AC*
2.3 *Calculate and set the join relationship(s)*
2.4 *For every nested PC that contains any selected element*
  2.4.1 *Create a new AC within this AC*
  2.4.2 *Call the recursive extraction function (parameters: this basePC, the new AC)*

Figure 10.1: The main feature unweaving algorithm and its recursive extraction function [Stoiber and Glinz, 2010a].

recursive extraction

The unweaving algorithm consists of five main steps (see the left-hand side of Figure 10.1). As a first step, the algorithm creates a base aspect container, which is the *feature container*. The feature container and the ADORA model as a whole are then handed over to the recursive extraction function to perform the extraction. This recursion works as described in the right-hand side of Figure 10.1. First, the so-called base parent container (the *basePC*, which is the lowest container element in the model's hierarchy that contains all the selected elements) needs to be found. After having the basePC found, all selected elements that are directly nested within this basePC container are extracted into the already created feature container, in the first recursive call (see the right-hand side of Figure 10.1). After this extraction, the necessary join relationships are computed and created, to restore the model's semantic equivalence (i.e., when this aspect would be composed again). Refer to [Jehle, 2010] for the ADORA-related details on creating these join relationships. As a last step of this first recursive call, for the case where more selected model elements exist more deeply nested in this basePC model element, a new part of a feature (i.e., nested AC) is created inside this recent AC and the next recursive call is performed. This next recursive call extracts the further selected model elements, using this basePC and the newly created AC as parameters. This way the recursion incrementally descends into the next lower part of the model, extracts all the selected elements, creates an adequate internal feature structure as well, and eventually terminates, when all selected elements are extracted.

finalizing the extraction

After the recursive extraction has terminated the main feature unweaving algorithm continues at step 3, see the left-hand side of Figure 10.1. The just refactored aspect-oriented ADORA model is now semantically equivalent to the original model—the content of the model is the same, but

only the structure is enhanced with additional aspect modeling. Because the recursive extraction part may have created redundant ACs (i.e., ACs that only contain other ACs but no requirements model elements) these will be removed again in step 3 (if present). After that, the final model is reached and the algorithm continues to fine-tune the automatically created layout of the just extracted feature in step 4. This layout optimization mostly reduces white space and coherently aligns the extracted elements. Finally, in step 5, the just extracted feature container is specified as variable, by setting its *variability* attribute to *true*. This recursively also turns all child ACs into variable ACs and produces a unique decision item that is annotated onto all outgoing join relationships that were created for this feature extraction. This newly created decision item will henceforth also be displayed in the decision table view.

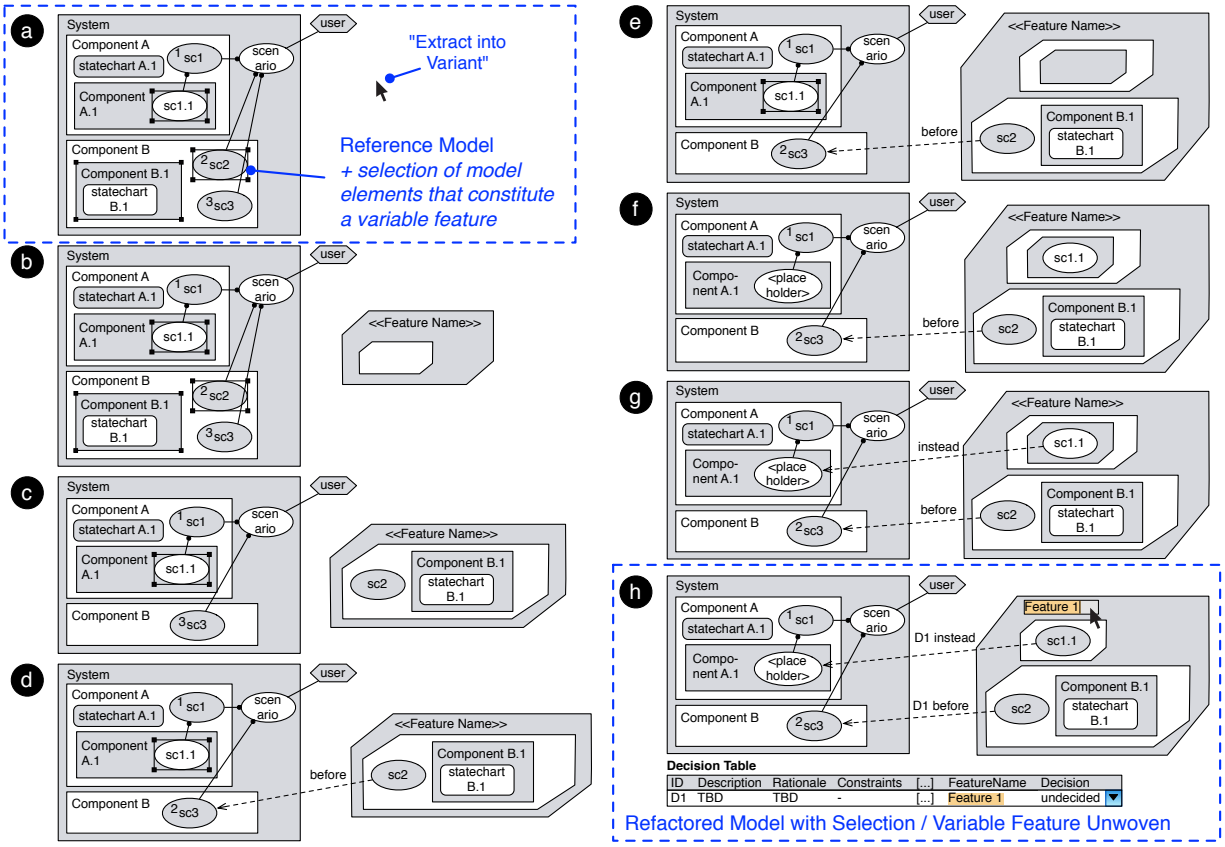


Figure 10.2: An illustration of feature unweaving; *a)* shows the original model with a variable feature selected, *h)* the resulting model with the selected feature unwoven, and *b-g)* show intermediate steps of the extraction [Stoiber and Glinz, 2010a].

illustration by example

Figure 10.2 briefly demonstrates how the feature unweaving algorithm performs a feature extraction in a concrete example. Diagram *a* in Figure 10.2 shows a non-trivial reference model, written in ADORA, where a product line requirements engineer has already selected the variable model elements that he or she wants to extract as a variable feature. The mouse pointer indicates the location where feature unweaving shall extract the feature to, in the graphic layout. The selection represents a heterogeneously cross-cutting variable feature—it involves model elements in several components of the system. Starting from this diagram, the subsequent diagrams *b-g* show various intermediate steps of the extraction process. The last diagram *h* shows the final model, with the whole feature unweaving algorithm processed.

An interesting peculiarity of this examples is that the scenario *sc1.1* needs to be replaced by a *«placeholder»* scenario element, when it gets extracted. Otherwise, there would not be any valid "hook" (i.e., join point or target model element of the join relationship that needs to be created) left to define the required weaving semantics. Also, the object *Component B.1* and the scenario *sc2* are extracted together because they are both in the same parent container. This requires an adaption of the sequence number of the scenario *sc3* from *'3'* to *'2'* and only one join relationship that connects these two scenarios. The weaving semantics used in this example is fully compliant with Meier's original definition for ADORA [Meier, 2009]. For a detailed description of feature unweaving please refer to [Jehle, 2010] or [Stoiber and Glinz, 2010a]. For a more general introduction, based on a different example, please refer to [Stoiber et al., 2010].

correct-by-construction variability refactoring

A combination of feature weaving and feature unweaving also allows an *undo* function for any already extracted feature. This operation also re-constructs the original selection, which is equivalent to all just woven elements. We call such an undo operation *feature removing*. This operation binds the associated decision item to $true$, but it performs this binding permanently within the product line model as a whole. Thus, it provides tool support to easily remove a variable feature's specification by weaving it and deleting its aspect-oriented modeling with only one click (Figure 11.2 for how this is integrated with the ADORA tool). This operation re-constructs the originally provided selection of model elements, which was extracted by feature unweaving. Even when the variable feature was manually created, then all of this feature's model elements will be selected after its removal. This selection can then easily be adapted and re-extracted into a changed variable feature specification, if desired. Such tool support allows evolving a SPREBA variability model with only minimal manual efforts, by using a combination of feature weaving (Chapter 7) and feature unweaving, as presented in this chapter. Importantly, such a tool-supported refactoring also assures that the underlying requirements model and aspect weaving semantics stay syntactically and semantically equivalent (i.e., correct) throughout such a product line evolution process. This combined concept eventually solves—to a significant extent—the problem of high specification, maintenance, and evolution efforts, as highlighted in Section 4.2. A validation has been presented in [Stoiber and Glinz, 2010a] and will also be discussed in the Chapters 12 and 14.

Part III

Empirical Evaluation

An Overview of SPREBA's Empirical Evaluation

The empirical validation of SPREBA's contribution, as presented in this thesis, consists of the following:

Constructive: First of all, a constructive validation is presented by an implementation of the SPREBA approach in the ADORA tool. This implementation provides a constructive validation and proves that it is possible to implement the presented concepts with a tool, see Chapter 11.

Feasibility: Secondly, the feasibility of SPREBA on basis of the ADORA modeling language has been validated by specifying several product line examples. These examples are from different domains (e.g., ranging industry to banking), are on very different levels of abstraction (from very abstract to rather detailed specifications), have been modeled by different people (e.g., students, researchers, professionals), have partially been compared to other modeling approaches (e.g., to UML together with OVM or feature modeling), and include hypothetical as well as real-world examples. Chapter 12 provides more details and a brief practical feasibility discussion for ADORA and SPREBA.

Performance: Thirdly, Chapter 13 provides a preliminary performance evaluation of SPREBA's SAT solving-based automated variability analysis. The evaluation is preliminary because it does not evaluate all operations as defined in Chapter 9, but focuses on SAT checks of domain models (i.e., of $\Phi(\mathcal{M})$), which is the most crucial operation. First, three different types of domain models get generated: (I) simple feature diagram-like models (without cross-cutting), (II) SPREBA models (with cross-cutting and weak dependencies), and (III) NP-hard CNF formulae with a critical clauses-to-variables ratio. These three types of models are then scaled up in size and the time needed to verify the model's satisfiability is plotted. The results show that SAT-based automated analysis of SPREBA models is about as feasible as it is for classic feature models. The results also show that both feature models and SPREBA models are clearly well-behaved and tractable SAT problems, compared to type (III) models, which scale much worse.

Case Study: Finally, Chapter 14 presents a case study within a company where SPREBA was applied and evaluated in comparison to a UML and feature modeling approach, to support the customization of enterprise resource planning (ERP) system solutions. Concretely, the ADORA tool and Sparx Systems Enterprise Architect along with pure-systems pure::variants were used to model the functionality and variability of Microsoft's Dynamics AX ERP system. A Goal Question Metric (GQM) catalogue was derived from research questions and typical product line-related activities. This catalogue was then systematically evaluated for the two product line requirements modeling approaches, both quantitatively and qualitatively. The results present concrete data and empirical findings about the particular strengths and weaknesses of the state of the art and the ADORA and SPREBA approach. Overall, considerable benefits for SPREBA were found, despite the fact that SPREBA's tool support is far from industrial-strength.

CHAPTER 11

Tool Implementation

Besides SPREBA's novel language design, as presented in the Chapters 5 and 6, the new concepts presented in this thesis (i.e., Chapters 7-10) are mostly novel conceptual solutions for tool support. The implementation of these concepts in a working tool, thus, constitutes an important part of their validation. This chapter provides a short overview on SPREBA's implementation in the ADORA tool.

ADORA *tool*

Conventional ADORA modeling and its view generation capabilities exists in a tool since [Seybold et al., 2003]. An improved implementation was also presented in [Reinhard et al., 2008]. [Meier, 2009, Chapter 11] provides a comprehensive and technical description of the ADORA tool's implementation, which is currently based on the Eclipse platform. [Meier, 2009, Section 11.4] also provides a very brief description of ADORA's aspect weaving implementation. The tool implementation presented in the following has been developed upon this existing, prototypical implementation. This chapter primarily focuses on the integration of SPREBA into ADORA's graphical user interface, however. For a detailed description of the implementation please refer to [Meier, 2009], [Kandrical, 2009], and [Jehle, 2010].

product line model creation

The first tool-supported activity required in the ADORA tool, when engaging into software product line modeling, is *feature unweaving*, which allows to automatically extract a selection of model elements into a variable feature (Chapter 10). Figure 11.1 shows how the feature unweaving plug-in is integrated with the ADORA tool's graphical user interface (GUI), as developed by Jehle [Jehle, 2010]. This Figure shows the graphic requirements reference model of a product line of industrial automation devices (as already used in Section 8.3), a selection of

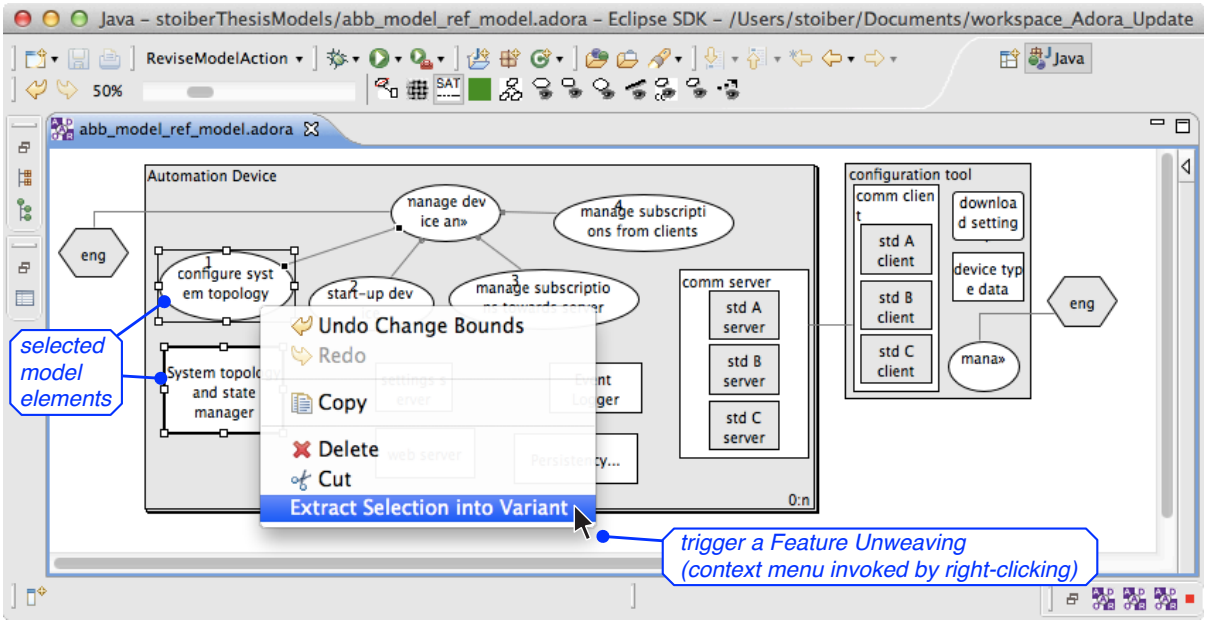


Figure 11.1: An example reference model in the ADORA tool, with a model element selection and the context menu opened to start a *feature unweaving* operation.

the variable model elements that constitute the variable feature *supervisory unit* and the opened context menu and action to perform the feature unweaving. When a user clicks on the action *Extract Selection into Variant*, then the mouse cursor changes to a crosshair and the user has to determine a graphic location in the ADORA model, where he wants this new feature to be extracted to, as illustrated in Figure 10.2. After that the remaining extraction will be performed fully automatically by the feature unweaving plug-in.

product line domain modeling

For existing SPREBA product line models the ADORA tool offers various view generation and automated analysis capabilities. Figure 11.2 shows a screenshot of the ADORA tool that presents the complete industrial automation devices product line, as already presented in Figure 8.7. For such a model the ADORA tool offers several view generation and automated analysis capabilities.

ADORA's explosive zooming and its view generation capabilities are highlighted on the top-left and top-right of Figure 11.2. In the top-middle area the function for *removing an already extracted variable feature* is highlighted, which weaves all the variable model elements of a selected feature, deletes the feature, and leaves all woven elements selected. This selection can then be adjusted manually and be re-extracted as a changed variable feature with feature unweaving, which allows a very simple domain model evolution by refactoring. Further, a toggle-button for enabling or disabling the continuous SAT-based analysis of the domain model

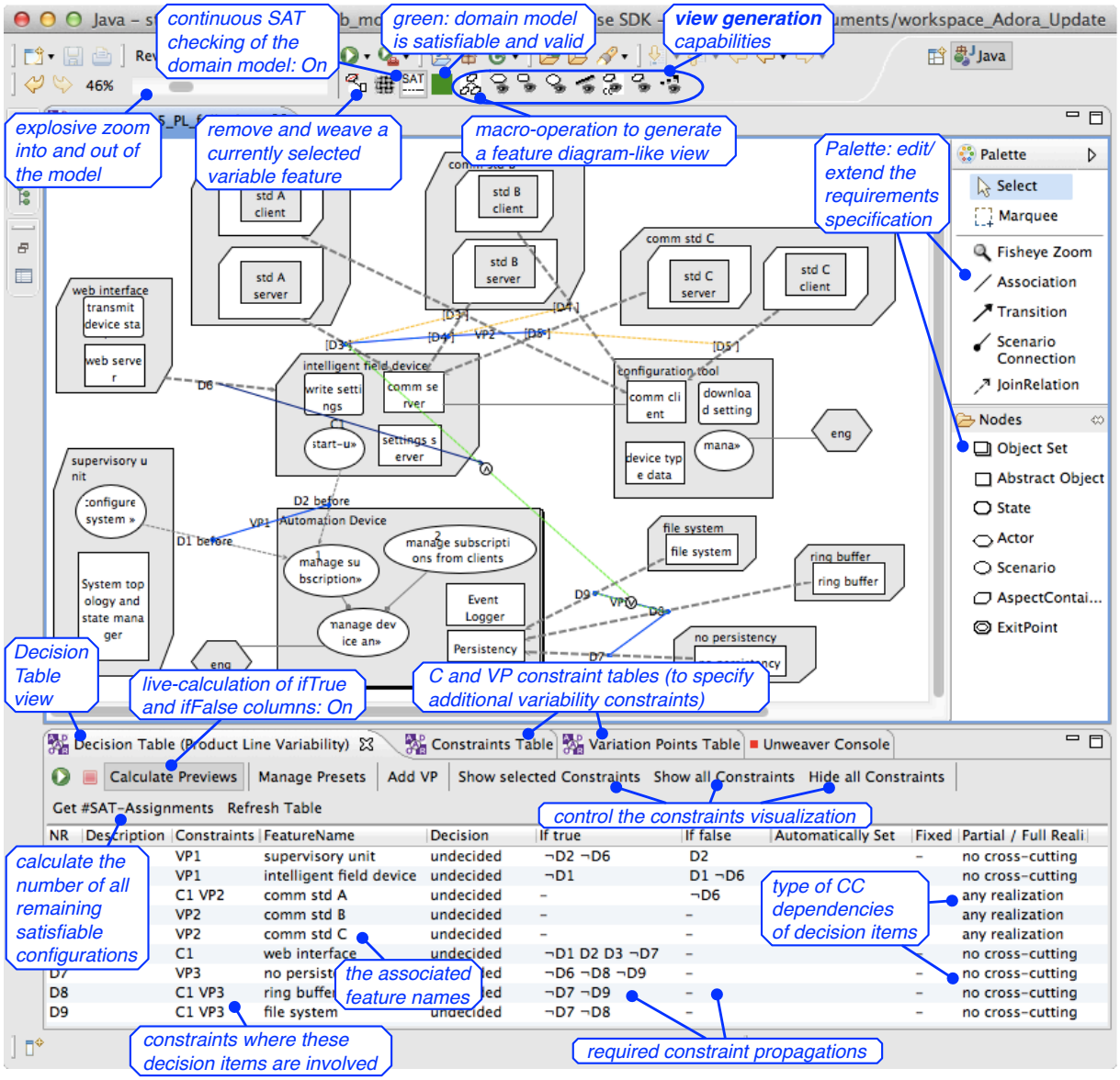


Figure 11.2: An example domain model in the ADORA tool, with the available information and operations highlighted in blue color in the GUI.

is shown. Whenever this button is *green* the model is fully satisfiable and no dead or mandatory decision items exist. When the continuous SAT-based automated analysis is not active and the variability model changes, then the button turns *orange*, which indicates an unverified state. When this orange button is clicked, then the model is verified and the color turns either to *green* (when everything is valid) or *red* (when dead or mandatory decision items exist). Further-

more, the button that creates a feature diagram-like view of the whole ADORA and SPREBA model is highlighted as well. This button essentially triggers a macro-operation that collapses all feature nodes and all commonalities with ADORA's fisheye zoom and also visualizes all constraints. This functionality is particularly useful to keep an overview of very detailed ADORA and SPREBA models.
In the middle of Figure 11.2 the actual ADORA and SPREBA requirements model is shown. In the middle-right the palette is highlighted, which is used to manually add new model elements and to edit the requirements model.
At the bottom part of Figure 11.2 the decision table view is shown, along with the tabs to open the variation points and constraints table views. The decision table view lists all decision items and their more detailed information, like the associated feature names, the constraints where they are involved, the required constraint propagations for a selection and deselection, and the cross-cutting dependencies. These are highlighted in Figure 11.2. Further, this view also offers a toggle button to disable or enable the automated calculation of all constraint propagations, every time the model changes. For very large models it may be advisable to disable this capability, for performance reasons. When this capability is disabled, specific single constraint propagations can still be calculated on demand, by double-clicking the field *if true* for any specific decision item, for example. This ad-hoc calculates and displays the constraint propagations that would be required for this decision. Further, the button for calculating the number of all products that are still satisfiable for the current configuration (the button *Get #SAT-Assignments*) and the buttons to control the visualization of variability constraints in the graphic requirements model, are highlighted. The variability constraints visualization is not intelligently routed, like associations are in an ADORA model, for example, but rather visualized with straight lines on top of the model. This makes the requirements model less readable, but makes the dependencies between variable features easier to comprehend. Because of the significantly increased model complexity, when all variability constraints are visualized, these are visualized by default, but rather can be displayed on demand via these buttons. For any decision item of interest, hence, the ADORA tool can straightforwardly visualize any variability constraints of interest, by clicking one of these buttons.

product derivation

Figure 11.3 further shows a screenshot of the ADORA tool during a stepwise, incremental product derivation. Four variability binding decisions have already been taken in this model: the *communication standards A* and *B* were selected and the *comm std C* and persistency option *no persistency* were deselected. The graphic ADORA and SPREBA model visualized in Figure 11.3, thus, already shows a partially derived model, with all requirements associated to the decision items set $true$ woven and with those set $false$ hidden, as specified in the Chapters 7 and 8. The variability constraints C1 and VP3 are also visualized in a derived form in this view, as a consequence.
At the bottom of Figure 11.3 the current configuration and the required constraint propagations for this particular configuration are shown in the decision table view. As highlighted by the drop-down list at decision item D8, the actual variability binding decisions are taken in the de-

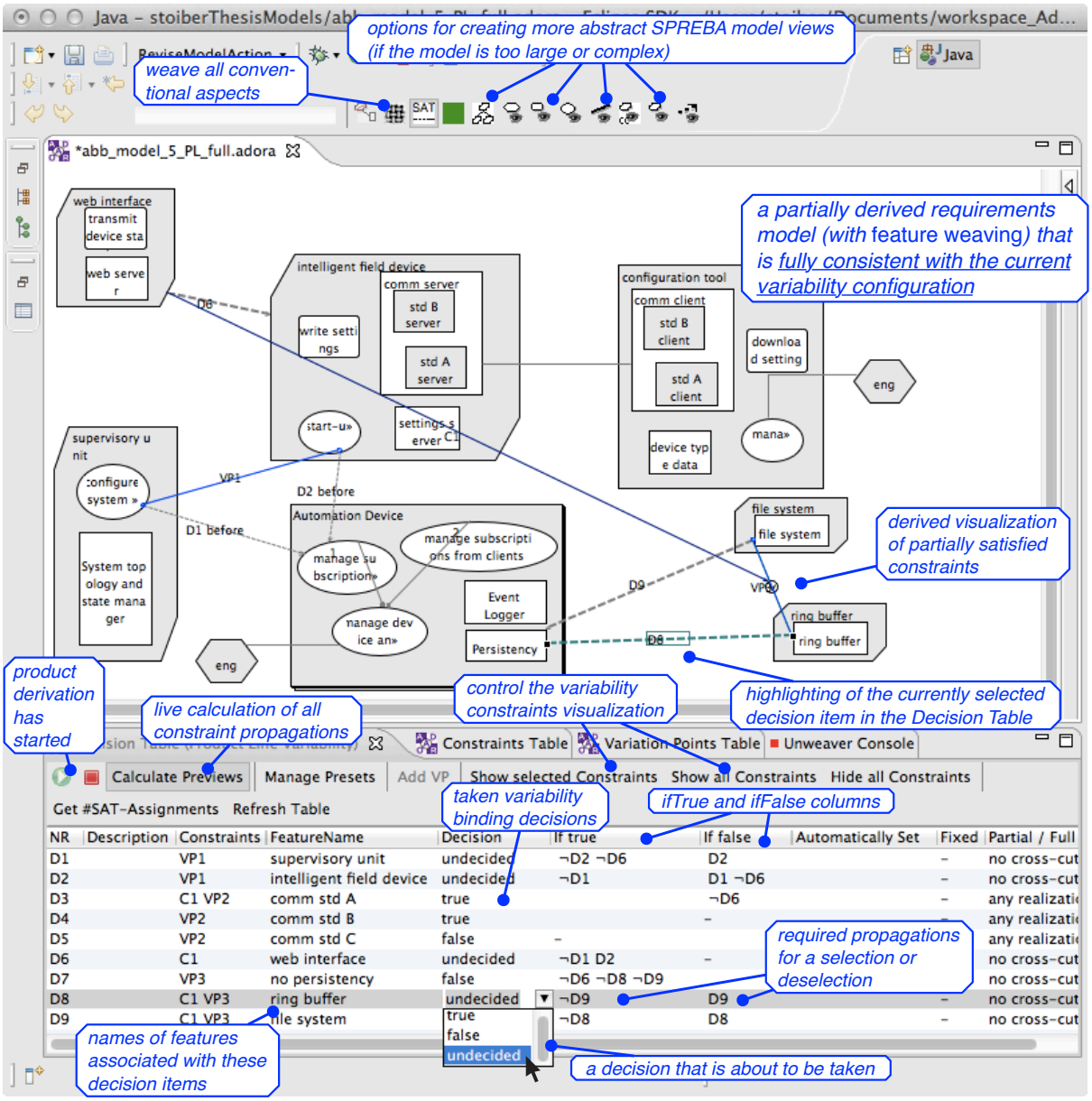


Figure 11.3: An example product derivation in the ADORA tool, with some variability binding decisions already taken and the important information and operations highlighted in blue color in the GUI.

cision table view (i.e., in this version of the tool; one could also implement a version where these decision are directly taken in the graphic model). Whenever a new variability binding decision is taken ($undecided \rightarrow true|false$) or an already taken variability binding decision is

changed ($true|false \rightarrow undecided|true|false$), also the required constraint propagations will immediately be set, as introduced in Chapter 8. Next to the decision column these constraint propagations are already shown as previews for the required consequences of either choice. For every decision item that is selected in the decision table view, all join relationships with which this decision item is associated are highlighted in the graphic model. This is shown for decision item D8, in Figure 11.3. Such a highlighting of join relationships allows a better navigation of the graphic model while reasoning about constraint propagations in the decision table view, as introduced in [Stoiber et al., 2008]. On the top-left, in the decision table view, it is indicated that a product derivation is currently ongoing, as the 'play' button is shown as pressed. Therefore, the feature weaving function and the required SAT-based automated analysis are already active in this state of the tool.
As shown on the top of Figure 11.3, ADORA's view generation capabilities can also be used during a product derivation, as with any conventional ADORA model. Even conventional aspects (i.e., cross-cutting concerns modeled with aspects that are not variable) can still freely be displayed in a woven or unwoven view, as desired. Also, a horizontal abstraction that filters only conventional aspects, for example, is still possible in the ADORA tool, even during a product derivation. This is indicated by the still enabled view generation buttons in Figure 11.3.

More technical details about ADORA's implementation of *dynamic weaving* are provided in [Kandrical, 2009] and more details about the implementation of *feature unweaving* are presented in [Jehle, 2010]. More details about the ADORA tool in general are provided in [Meier, 2009]. ADORA's SAT-based automated analysis, as presented in Chapter 9, uses SAT4J[1]—a popular and speedy SAT solver that is also available within the Eclipse platform—and is implemented as specified in Chapter 9.

[1] See http://www.sat4j.org/ (checked on July 5, 2012)

CHAPTER 12

Feasibility Evaluation of ADORA and SPREBA

weak ADORA *validation*

There is only little empirical validation of the ADORA modeling language and approach in comparison to other state-of-the-art languages and tools. The only systematic validation of ADORA that also explicitly addressed another state-of-the-art language was presented in [Berner et al., 1999], where ADORA was evaluated against the UML 1.3. This validation was rigorous, but the UML has been revised significantly since then. Today, hence, we do in fact not know whether the ADORA language can still compete with recent languages like the UML 2.4 and with modern tools, like Sparx System's Enterprise Architect, for example. Moreover, the software industry has not yet adopted ADORA modeling. Hence, it is unknown whether plain ADORA (without variability modeling) even scales as well as the UML does for real-world industry projects. The personal impressions from [Zoller, 2010] and [Schadauer, 2011] actually suggested that plain UML with Sparx System's Enterprise Architect performs better in handling a model's complexity than ADORA does in the current tool prototype. The reason for this could in part also be the ADORA tool's low maturity. The ADORA tool has been developed with very limited funding, compared to major industrial-strength tools like the Enterprise Architect or an IBM Rational tool. A recent systematic and comparative study has not been performed since [Berner et al., 1999], however. This lack of empirical data for ADORA made it difficult to rigorously validate the SPREBA approach with a realization and implementation based on ADORA.

experience with ADORA *and SPREBA*

However, the modeling of several product line examples from different domains, as follows, has shown that (i) ADORA and SPREBA are feasible for model-based product line specification, (ii) that the expressiveness of ADORA and SPREBA is comparable to using the UML and OVM,

when modeling functional product line specifications, and (iii) that the SPREBA approach is also feasible for more efficiently handling requirements reviews when planning very large systems. The latter did not build on the ADORA tool, though, but only on using SPREBA's ideas to structure a text-based requirements specification by explicitly specifying features, as follows in Section 12.3. Recently, SPREBA has also been applied to support the release planning of a media management solution, again based on a textual requirements specification [Fricker and Schumacher, 2011]. This project has successfully used the SPREBA variability modeling concepts and feature unweaving in particular, to specify variability in a product management context with a text-based requirements specification. This study provided additional evidence for the feasibility of SPREBA, when applied to the areas of software product management and release planning, see [Fricker and Schumacher, 2011], similarly to the study described in Section 12.3. However, the upcoming Sections 12.1 and 12.2 first present an evaluation of the general feasibility of SPREBA based on ADORA.

12.1 Applying ADORA and SPREBA

In the early proof-of-concept phases for SPREBA's basic ideas a focus was laid on evaluating the general feasibility of aspects for modeling the variability in an integrated requirements specification. Therefore, a hypothetical and rather detailed product line example was studied and modeled. To complement these initial results a quite abstract, high-level requirements model of a large real-world industrial product line was further studied.

12.1.1 Electronic Home Security System Example

This hypothetical example illustrates the specification of the functionality of a door unlocking component in an electronic home security system. The example was chosen because it is easy to imagine and because of its similarity to the already established home automation example used in [Pohl et al., 2005].

The model was created by the author of this thesis, for illustration purposes and as an initial example to evaluate the feasibility of using ADORA aspects for variability modeling. The semi-formal ADORA requirements model and its integrated aspect-oriented variability model can be found in [Stoiber et al., 2007, Figure 1]. The model has been used in two publications as a running example. First, in [Stoiber et al., 2007], to demonstrate SPREBA's basic idea of modeling variability with aspects. Second, in [Stoiber et al., 2008], to demonstrate how ADORA's view generation capabilities can be also used on SPREBA models and how they can provide advanced product line model visualization support.

The core results gained are a proof-of-concept of the approach in the form of evidence that ADORA aspects are technically feasible for modeling variability in ADORA models.

12.1.2 Industrial Automation Devices Exemplar

This example was created in cooperation with a global Fortune-500 company with a tradition in developing industrial software-intensive systems. The model has been created in a research effort that followed a previous study—see [Fricker and Stoiber, 2008]—which identified that the company does not maintain a company-wide product line variability specification. Much of the understanding of commonality, variability and variability interdependencies between products was tacit and distributed in the organization. Variability-related tasks were not addressed explicitly, but rather negotiated as part of the requirements reviewing activities. This lead to numerous reviews that strongly relied on personal knowledge and the expertise of individuals, down- and upwards the composition hierarchy. The created product line model demonstrated—on a very high level of abstraction—how an explicit company-wide variability specification could be realized.

The product line model was created by two senior domain experts in cooperation with the author of this thesis. Figure 8.7 already showed this high-level product line requirements model. The model has been used as an example in three publications: (1) to illustrate how the SPREBA approach allows an explicit specification of crucial product line variability knowledge that has yet not been documented explicitly [Stoiber and Glinz, 2009], (2) to demonstrate SPREBA's stepwise, incremental product derivation capabilities, which seamlessly integrates configuration and product generation, as also demonstrated in Section 8.3 [Stoiber and Glinz, 2010b], and (3) to demonstrate SPREBA's feature unweaving capabilities as a means to strongly improve the efficiency in variability creation [Stoiber et al., 2010].

As a result, our experience from this real-world industrial exemplar has shown that the application of SPREBA on high abstraction levels and in a real-world industrial context is principally and technically feasible. Jehle's work has also confirmed these results [Jehle, 2010].

12.2 Comparing SPREBA with State-of-the-Art Approaches

Besides this pure feasibility evaluation a comparative evaluation has been performed as well. The following two software product line examples were modeled with both ADORA and SPREBA and with state-of-the-art approaches, once with the UML and OVM and once with the UML and feature modeling.

12.2.1 Point-of-Sales System Example

The point-of-sales (POS) system example is a hypothetical example that was created in a purely academic environment. A simple point-of-sales system requirements specification was modeled both with the UML and OVM and ADORA and SPREBA. Two roles existed: a customer stakeholder, who had to provide the requirements for several concrete point-of-sales systems, and a modeler, who elicited and modeled all the required functionality and variability in both the UML and the ADORA language. The customer stakeholder role was played by the author of this thesis and the modeler role was performed by a student who did his informatics specialization project. The modeler's task was to elicit all important requirements in interviews and to model these requirements and all the required variability in the two different languages. The tools used were Smartdraw 2010 for UML and OVM and the ADORA tool for ADORA and SPREBA.

The modeler came up with a comprehensive requirements model (i.e., a detailed architectural, behavioral and user interaction specification), three variation points and eight variants (i.e., variable features). The resulting models were all of a good quality and well readable for peers and examiners, in both approaches. The results were that ADORA and SPREBA were found feasible and as expressive as the UML and OVM notations. The student argued that the ADORA and SPREBA model gave a slightly better overview over the variability specification, however, because the model was integrated. The used example was rather small, though, which limits the external validity of this study. The complete ADORA and SPREBA specification was still readable when printed on one page in A3 format.

The important result from this project was that ADORA and SPREBA modeling has been found as expressible as UML and OVM modeling, with slight benefits for ADORA and SPREBA for overview and understandability of the product line model.

12.2.2 GoPhone Case Study

GoPhone is a hypothetical software product line in the mobile phone domain, developed by the Fraunhofer Institute for Experimental Software Engineering (IESE) and documented in detail over more than hundred pages, see [Muthig et al., 2004]. This case study specifies a typical software product line exemplar (i.e., in the mobile phone domain) and was originally also intended to serve the community as a common example to allow a better comparison of different approaches to software product line engineering.

A bachelor's thesis project has been arranged to systematically evaluate the feasibility of the UML and feature modeling and ADORA and SPREBA, by modeling the GoPhone software product line [Zoller, 2010]. Sparx Systems' Enterprise Architect was used for UML modeling, in combination with pure-systems' pure::variants for feature modeling. These are two major state-of-the-art and state-of-the-practice tools in this area. The ADORA tool was used for

ADORA and SPREBA modeling. The main focus was laid on a systematic comparison of the underlying concepts.

The bachelor student has created the models at exactly the same level of detail in both approaches, as specified in [Muthig et al., 2004], with only minor assumptions where required information was missing. Together with two academic advisors (one of them the author of this thesis) special care was taken that the actual models in both approaches were equivalent in content and semantics.

Zoller's modeling and qualitative analysis has shown that ADORA and SPREBA were expressible enough to allow specifying all the requirements and variability, as also specified with Enterprise Architect and pure::variants (EA+p::v, abbreviated) [Zoller, 2010]. All information about requirements and variability was reasonably expressible and the two created product line models were found as semantically equivalent. No significant shortcomings in either modeling language were identified.

In his quantitative analysis, Zoller has found that overall significantly less model elements were required to specify the same functional requirements with ADORA and SPREBA, compared to the UML and feature modeling [Zoller, 2010]. While the EA+p::v specification required an overall of 949 model elements (i.e., nodes, lines, and constraints), the ADORA and SPREBA model required merely 594, as aggregated in [Stoiber and Glinz, 2010a]. The reason for this was the orthogonality between the UML and feature modeling, which required a significant portion of redundancy to guarantee correct traceability between all diagrams in the UML-based approach. Section 12.4 analyzes this finding further. Overall, Zoller had to create seventeen dedicated diagrams for the UML specification plus one feature diagram, while one integrated diagram sufficed for ADORA and SPREBA [Zoller, 2010].

The major results of Zoller's study were (i) that relying on ADORA and SPREBA was fully feasible for a model-based specification of GoPhone's functional requirements and variability and (ii) that ADORA and SPREBA required about *60% less model elements* for the same requirements and variability specification, compared to the UML and feature modeling with EA and p::v. This reduction of required model elements suggested that the ADORA and SPREBA paradigms also improve the efficiency of product line requirements specifications. A more detailed overview of these results can be found in [Stoiber and Glinz, 2010a].

12.3 Applying SPREBA to handle Variability in Requirements Relevancy

All the above examples were models of classic software product lines, where multiple heterogeneous products with a significant amount of common functionality were developed. However,

we found that modeling variability in requirements specifications can also be beneficial when developing large monolithic systems, which are *not* software product lines, see [Stoiber and Glinz, 2010a] or [Fricker and Schumacher, 2011]. This is especially the case when multiple stakeholder groups with significantly differing interests in the software system are involved. Modeling variability in such a context allows that some specification, which is required by all stakeholder groups, can easily be excluded from the specification and, hence, does not need to be reviewed by all these stakeholder groups. SPREBA has been applied during the requirements phase for such a monolithic system with rather complex stakeholder constellations.

12.3.1 A Governmental Decision Support System Exemplar

An early-phase requirements specification was created for a decision support system for a government in Europe. The nature of this software product was significantly different from the above cases, as it was not a product line of systems but merely a single, monolithic system. The requirements engineer in charge had to deal with a significant amount of variability of requirements between the involved stakeholder groups—some stakeholder groups had specific requirements that others did not share. SPREBA was adopted to handle this variability.

The requirements engineer who created the specification and applied the SPREBA approach was a senior researcher and industry consultant. An overview of the requirements specification and the created SPREBA model is also presented in [Stoiber and Glinz, 2010a, Section IV]. An interesting fact was that only very few parts of the specification were really relevant for all stakeholders (i.e., the commonality was very small). Many requirements could be grouped together to features, as they varied together between stakeholder groups. Compared to the industry's ad hoc approach of dealing with variability, which is using AND/OR tables (Section 3.2.1), SPREBA has lead to a profound improvement of the variability modeling efficiency, as previously reported in [Stoiber and Glinz, 2010a].

The qualitative results of this study were that SPREBA's application area is potentially wider than classic software product line engineering projects. The study has shown that SPREBA can also be successfully applied in conventional development projects, to increase the efficiency of requirements engineering processes when significant amounts of variability occur between stakeholders' or stakeholder groups' interests.
From a quantitative point of view, SPREBA and feature unweaving (Chapter 10) have been found to reduce the effort for all variability-related tasks by factors between 2.8 to 4.8, compared to the hitherto prevalent AND/OR table-based solution [Stoiber and Glinz, 2010a]. Interestingly, these results were already measured for only three product requirements specifications—more products would increase these benefits even more. Hence, the potential improvements when applying the SPREBA concepts instead of an ad hoc AND/OR table-based approach can in general be described as profound.

12.4 Discussion: Are ADORA Aspects feasible for Variability?

General knowledge about aspect-oriented modeling and variability modeling (recall Section 2.4 and Chapter 3) and the hitherto experience (see above) in modeling variability with ADORA aspects has led to the conclusion that aspects in ADORA are actually feasible for modeling variability. The reasons are that, firstly, both aspects and variability are essentially asymmetric concepts, when compared to the core model. Secondly, also ADORA's extended aspect weaving semantics has largely shown to be expressible enough to accurately model variability, as discussed above and as follows.

asymmetry of variability

The natural relationship between commonality and variability in software product lines is asymmetric. The product line's commonality is always present and therefore is naturally kept as a plain model. Variability, or variable features, on the other hand, typically represent additional functionality that can be added to the commonality, when desired. The commonality of a software product line is typically valid by itself (i.e., as far as allowed by the variability constraints) because it is typically modeled as a plain diagram that is self-contained and that is not intended as only a model fragment. Variable features, on the other hand, are often not valid by themselves and, hence, often not self-contained. This means that some variable feature's specifications simply may not make any sense without viewing it in the context of the specified join points. Features, hence, may depend on their parent features and the commonality to make sense, while the commonality typically already makes sense by itself. Therefore, variable features may be classified to have an asymmetric relationship to the commonality of a software product line (not necessarily in all cases, though).

asymmetry of aspects

The natural relationship between homogeneous cross-cutting concerns (i.e., classic *aspects*) and the core concerns in a software system is also an asymmetric one. Cross-cutting concerns require a severalfold and redundant specification of particular model fragments in different core concerns or in other cross-cutting concerns. Aspects are used to efficiently model these cross-cutting concerns. While the core concerns of a software system are typically self-contained, classic aspects typically model diagram fragments. Whittle and Jayaraman distinguished modern aspect-oriented modeling approaches into symmetric and asymmetric ones (the term 'asymmetric' was not mentioned literally, though) [Whittle and Jayaraman, 2007], recall Section 2.4.1. Classic aspect-oriented modeling approaches are asymmetric. Such an asymmetric approach is the more widely known AspectJ, for example. ADORA's aspect-oriented modeling solution [Meier, 2009] is also asymmetric, recall Section 2.4.2. This asymmetry of both concepts—variable features and classic aspects—makes aspect-oriented modeling a well-suited choice for variability modeling.

generalized ADORA aspects

ADORA's original aspect weaving semantics by [Meier, 2009] mainly dealt with cross-cutting behavior and scenario chunks, recall Section 2.4.2. In many cases, however, Meier's original aspect weaving semantics turned out to be too restrictive for modeling variable features. This was

especially the case in more abstract, higher-level models, as shown in Figure 8.7, for example. This issue has been addressed since ADORA's initial dynamic weaving capabilities were developed by [Kandrical, 2009], where ADORA's aspect-oriented modeling and weaving capabilities were extended with more general cases. For instance, the aspect-oriented modeling of abstract objects only (i.e., without any additional behavior or scenario chunks) and the composition of such a model was first realized in [Kandrical, 2009], which was not foreseen by [Meier, 2009]. This extended weaving semantics was later also revised and refined by [Jehle, 2010]. In early-phase models and in most real-world models, as discussed above, such abstract cases occurred quite often and a detailed specification of behavior chunks and scenario chunks was rather rare.

are ADORA *aspects suitable?*

Specifically in [Jehle, 2010] and [Zoller, 2010] a special focus was laid on evaluating the feasibility and expressiveness of using only ADORA aspects for modeling a system's variability. While Jehle has focused on the industrial automation devices (Section 12.1.2) and the governmental decision support system (Section 12.3.1) cases, Zoller has studied the GoPhone case (Section 12.2.2). For all these cases, which are quite heterogeneous in terms of industry, modeler, abstraction level, and size, the authors found that using ADORA's generalized aspect-oriented modeling was sufficiently expressible for variable features that occurred.

yes – types of variable features

An important result found in [Zoller, 2010] and also reported in [Stoiber and Glinz, 2010a] was that a majority of the variable features of the GoPhone case (i.e., 15 out of 25) were actually heterogeneously cross-cutting (i.e., their realization was scattered over multiple abstract objects and other features)—recall Chapter 10 for definitions of types of variable features. Purely local variable features were considerably less frequent (i.e., 9 out of 25), but did also exist in the GoPhone case. Homogeneously cross-cutting features (i.e., classic "aspects") were very rare (i.e., only one out of 25 variable features). We observed, however, that the amount of cross-cutting was strongly influenced by the level of detail of the model. Since the GoPhone case was also intended as a general example to the community [Muthig et al., 2004], these results are interesting. The findings show that classical aspects (i.e., homogeneously cross-cutting concerns) rarely occur as variable features, but that a majority of variable features are in fact heterogeneously cross-cutting over the system's structure (i.e., the system's primary and dominant decomposition). As ADORA's feature unweaving is particularly suitable for handling and specifying such heterogeneously cross-cutting variable features (Chapter 10) this makes ADORA aspects very well suitable for variability modeling.

no – limited weaving semantics

On the downside, however, Zoller was required to do numerous manual adaptions to the detailed behavioral specification (i.e., ADORA's statecharts) to be able to extract the variable behavior chunks into appropriate variable features [Zoller, 2010]. The reason for this was that ADORA's current weaving semantics allows a behavior chunk only to be woven into a single state transition. Thus, the behavior chunk as a whole can only have one incoming and one outgoing state transition—otherwise, it can not be modeled with an aspect. In most cases, the original behavioral specification of most variable features was tightly integrated into the overall behavioral modeling and in the majority of cases has several incoming and outgoing state transitions.

Hence, strictly taken, such behavior chunks could not be modeled with Meier's aspect weaving semantics [Meier, 2009] because they simply can not be specified and woven with a single aspect. Similarly, a selection of any such behavior chunk (i.e., a selection of states and state transitions that overall has more than one incoming and one outgoing JR) can also not be extracted with feature unweaving in ADORA (Chapter 10). However, it was always possible to add additional pseudo-states and to modify the behavioral specification such that single behavior chunks were extractable. This again allowed a decomposition of the system with Meier's aspects. These modifications were semantics-preserving, but bloated the model in size. Most of these pseudo-states were usually unnamed and the system remained only infinitesimally short in them, as no events were specified for a further transition. In the GoPhone reference model such behavioral modeling adaptions needed to be done quite frequently, when the reference model was evolved into a SPREBA product line model. Overall, the total number of states increased from 80 before the aspect-oriented variability modeling to 151 afterwards—an increase by 88.7%. This number, however, also includes all exit point states, which have to be defined in every aspectual behavior chunk specification, see [Meier, 2009]. Nevertheless, for the remaining parts of the model the aspect-oriented variability modeling went without difficulty. ADORA's asymmetric and AspectJ-like weaving semantics is still a limitation, however—in particular for detailed behavioral specifications.

conclusion: aspects are feasible

Variable features in a product line often cross-cut the commonality and/or other variable features. Such situations make aspect-oriented modeling an excellent choice because every variable feature that is cross-cutting can directly be specified as a cross-cutting feature straight away. This allows a natural variability specification and in turn even reduces the amount of additional variability constraints. Many dependencies between variable features further already specified by the features' hierarchies, cross-cutting, and dependency types (i.e., weak or strong decision item associations) and do not need to be added additionally. Also, when heterogeneously cross-cutting variability occurs (i.e., classic 'aspects', which are rare but yet do exist), aspects are ideal, as they allow a better maintenance and evolution of such concerns [Meier et al., 2007].
On the other hand, when extracting behavior chunks of already modeled behavioral specifications ADORA's aspect weaving semantics is limited, as described above. Furthermore, for all variable features we dealt with, a consistent and precise model adaption and aspect-oriented modeling was eventually always found, with still manageable extra effort. To alleviate these expressibility issues, nevertheless, we suggest to investigate the expressiveness of ADORA's aspect-oriented behavioral modeling capabilities more closely in the future research (Section 15.2).

Overall, we conclude that aspects are indeed feasible and well-suited for modeling variable features in an integrated requirements model. Future applications of SPREBA could even build on even more powerful aspect-oriented approaches like MATA [Whittle et al., 2009] (recall Section 2.4.1), which would provide a superior expressibility. From a modeler's point of view, however, ADORA's present aspect-oriented modeling capabilities were already fully capable of

precisely specifying any variable feature that was observed in any of the studied product line exemplars.

CHAPTER 13

Performance Evaluation: SAT-based Constraints Analysis

SPREBA models allow a more fine-grained variability modeling than feature models and, thus, also require a more rigorous automated analysis, recall Chapter 9. This chapter presents a basic empirical analysis of three different types of variability models: (I) feature diagram-like models (i.e., without cross-cutting), (II) rather complex SPREBA models (i.e., that include significant amounts of cross-cutting and weak dependencies), and (III) random 3-CNF formulae with a critical clauses-to-variables ratio (of 4.1). Models of the type I provide a good comparison to state-of-the-art variability models. Models of type II are used to show how computation-intensive SPREBA models are. Models of type III provide empirical data about particularly hard problems. Earlier results reported in [Mendonca et al., 2009] argued that SAT-based automated analysis of feature models is 'easy', which means that their SAT-based automated analysis is *well-behaved* and tractable. However, Mendonca et al.'s empirical results were largely based on unsatisfiable models and, hence, did not really prove this general conclusion. The empirical evaluation presented in this Chapter is based on only satisfiable models. Therefore, it provides significantly more evidence that both feature models and SPREBA models are indeed well-behaved, when analyzed with a SAT solver.

goal

The goal of this chapter is to evaluate models of type I (feature models) and type II (SPREBA models) and to find out whether they are well-behaved when analyzed with a SAT solver. Showing that this is the case provides a strong validation for the feasibility of this SAT-based automated analysis. Models of type III are by definition not well-behaved, but are also included to reject the null hypothesis (i.e., that all generatable satisfiable models are well-behaved).

related work There are similar empirical results for feature modeling—most importantly [Mendonca et al., 2009], as mentioned. Mendonca et al. random-generated feature models (where every feature has between 2 and 10 child features) and added random-generated cross-tree constraints. For these generated cross-tree constraints they used a cross-tree constraint ratio (CTCR) of 30%, which means that 30% of all features were involved in a constraint (and that the remaining 70% were not). These cross-tree constraints were purely random 3-CNF formulae with clause densities between 0.1 to 3.5. These also lead to conflicts, however, and caused many feature models to be *unsatisfiability*. [Mendonca et al., 2009, Figure 3] shows this very well, where even for small clause densities some models were already unsatisfiable. For high clause densities nearly all of them were unsatisfiable. Eventually, Mendonca et al. concluded that SAT-based automated analysis scales well for realistic feature models with up to $10'000$ features. This general conclusion is not fully justified, however. In particular, the use of unsatisfiable models in Mendonca et al.'s experimental design caused the following two major problems.

First, the larger Mendonca et al.'s models became and the larger the random-generated constraints became, the more contradictions existed among the variability constraints (e.g., as shown in [Mendonca et al., 2009, Figure 3]). As a SAT solver only needs to identify a single contradiction to prove the unsatisfiability of a model, this leads to the paradox result that models with even more constraints become even easier and faster to solve (see [Mendonca et al., 2009, Figure 4]). If these models were satisfiable at such a high clause density, as shown on the right-hand side of [Mendonca et al., 2009, Figure 4], then the required time for a SAT check would be *much* higher. Therefore, we consider Mendonca et al.'s general conclusion as not really proven by this experiment. Using satisfiable models instead of only unsatisfiable ones would most likely yield very different results.

Second, unsatisfiable models may make it very hard to do any meaningful automated analysis because *any* partial or full variability configuration will also be unsatisfiable. Hence, a SAT-based analysis similar to the one we introduced in Chapter 9 can not be realized with such models. The SPREBA approach explicitly verifies and maintains the satisfiability of every variability model at any time (Section 9.4.1)—any created SPREBA model is therefore satisfiable. This is not the case for Mendonca et al.'s results [Mendonca et al., 2009]. Thus, their conclusions can not be generalized to satisfiable feature models. And neither can they provide a reliable estimation for SPREBA models.

overview To gain actual empirical evidence about the hardness of satisfiable SPREBA models for SAT-based automated analysis we constructed an independent empirical performance evaluation. First, we planned the specification of models of the types I, II, and III for such an evaluation, as follows in Section 13.1. Second, we generated these models in an additional plug-in in the ADORA tool, ran SAT checks, and recorded and plotted them for random-generated models with increasing numbers of decision items, as presented in Section 13.2. Importantly, all models generated and used for this data were *satisfiable*.

13.1 Generating Models

To perform this empirical analysis, models of the mentioned three types had to be generated. For all models of the types I and II exactly one decision item has been used for every feature. Hence, the terms feature and decision item are synonymous in this chapter (whether parts of features with different decision items are realized does not make really a difference for the SAT-based automated analysis, but rather poses a challenge for feature weaving). For all generated models only empty abstract objects and empty variable aspect containers were used (as only the variability model's automated analysis was evaluated and feature weaving is out of scope). The following types of models were generated and used.

types I, II and III specifications

Type I. These models are classic *feature diagram*-like models. They were created in a pure tree structure, where every variable feature has exactly one parent feature. All child-parent dependencies, hence, are strong dependencies. 20% of all created features directly impact the commonality (i.e., the root feature). Every variable feature has between 0 and 5 child features (the actual number is randomly chosen).

Type II. These models are typical *SPREBA* models. 20% of all features again directly impact the commonality. Every second feature has between 1 and 3 parts of a feature (nested ACs)—the number was randomly chosen. Every feature and every part of a feature (variable AC or nested variable ACs) has one outgoing JR. All JRs that belong to a specific feature (including the parts of a feature) have the same unique decision item associated, as mentioned. Every tenth AC (either a feature or a part of a feature, randomly chosen) has either 1 or 2 additional outgoing JRs (this causes additional cross-cutting). Overall, 50% of all variable join relationships have their decision item only weakly associated with them (i.e., as weak dependencies, recall sections 6.1.1 and 9.3), while the other 50% have their decision item strongly associated (i.e., strong dependencies). The variable JRs with weak dependencies were randomly chosen.

These models of the types I and II were evaluated without and with VP and C constraints. We did so to also evaluate the impact of these additional constraints on the SAT-based automated analysis performance. VP and C constraints were added as follows.

Constrainedness by VPs. Variation point (VP) constraints (i.e., cardinality-based constraints) were considered only for child features of a specific feature (where these child features could also be cross-cutting). A VP constraint, hence, is possible wherever the incoming JRs to a variable feature have two or more different decision items associated with them. This number of possible VP constraints is called m and defines the upper-bound of realizable constraints. For every specific model of type I or II, hence, m was calculated. Based on this number m different degrees of constrainedness by VP constraints then were realized: 0%, 25%, 50%, 75%, and 100%, where 0% means that no VP constraints are in the model and 100% means that m VP constraints are in the model.

At every possible location for a VP constraint there is a particular number of child decision items x, which is at least two. A random number between 2 and x was chosen to determine how many of these decision items are part of the VP constraint. The min and max cardinalities of a VP constraint then were also randomly chosen, within the allowable bounds. For every new VP constraint added we verified that the model was still satisfiable (which is verified by a simple SAT check, i.e., SAT($\Phi(\mathcal{M})$), recall Chapter 9). If not, the constraint was re-generated until a satisfiable definition was found.

Constrainedness by Cs. Arbitrary Boolean logic (C) constraints were randomly generated as well, in a rather complex form. Every antecedent and consequent term of a C constraint consists of two 2-CNF clauses with randomly chosen decision items. The chosen operator was always an implication. An example of such a random-generated C constraint would be $C15 : ((D168 \vee D650) \wedge (D112 \vee D542)) \Rightarrow ((\neg D193 \vee D912) \wedge (D150 \vee D177))$. We chose this rather complex form of C constraints to generate examples that are at least as hard or harder than real-world examples (i.e., where often only requires and excludes dependencies are used as constraints between decision items). This allows a better external validity of the results, where one can argue that the performance of real-world models will most likely be within the found boundaries, or faster.
For simplicity, the number of maximally possible C constraints that were added to a model was chosen to be equivalent to the number of maximally possible VP constraints, namely m. Based on this number m the constrainedness by C constraints was also realized as 0%, 25%, 50%, 75%, and 100%, where 0% means no C constraints are in the model and 100% means that m C constraints are specified. Again, every time a C constraint was random-generated it was only added to the model when the overall model was still satisfiable afterwards (which is verified after every relevant model edit operation by a simple SAT check of $\Phi(\mathcal{M})$).

Type III. These models are mere random-generated 3-CNF formulae with a critical clauses-to-variables ratio. The chosen clauses-to-variables ratio for models of this type was 4.1.

hardware and software used

The hardware on which these models were generated, evaluated and plotted was a 2.00GHz Intel(R) Xeon(R) E5405 server machine, with 4GB of RAM. The software used was Microsoft Windows Server 2003 R2 x64 SP2 OS and Eclipse for RCP/Plug-in Developers with SAT4J (http://www.sat4j.org/) as a SAT solver.

13.2 Results

The generated models of the types I and II started from a size of 30 decision items. Their size was then increased (with $+1$ increments) until $1'300$ decision items for models of type I and until 930 decision items for models of type II. For every size, three different categories of models were generated: ones with only VP constraints, ones with only C constraints, and ones with

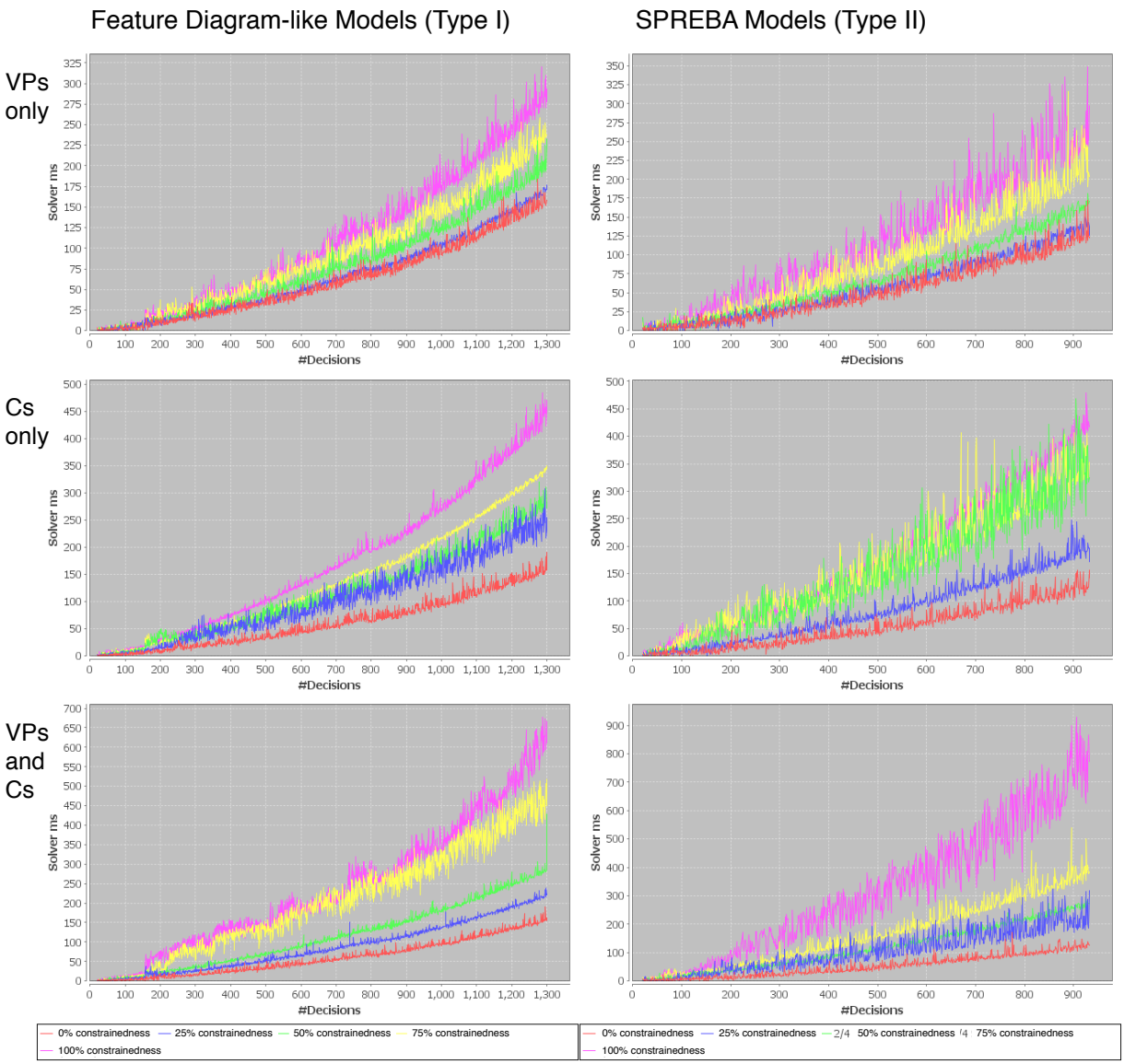


Figure 13.1: The aggregated SAT performance results of SPREBA models of the type I (left) and type II (right); three different categories were evaluated for every type: VP constraints only (top), C constraints only (middle), and VP and C constraints (bottom); and for every of these categories different amounts of constrainedness were evaluated: 0%, 25%, 50%, 75%, and 100% (indicated by the different colors); the x-axes show the number of decision items and the y-axes the time needed for SAT solving in milliseconds.

both VP and C constraints. Additionally, for the different categories of constrainedness (i.e., 0%, 25%, 50%, 75%, and 100%) the models were separately analyzed and plotted. This allowed

us to see the impact of additional constraints to the overall required time for a SAT check. To reduce the noise in the resulting plots, ten (10) dedicated models were generated for every size (i.e., # of decision items). Every data point, as follows, is an average run time for the SAT check out of ten models, hence. We chose average values instead of median values because they better reflect the occurrence of outliers, which we may expect when analyzing potentially intractable problems—the more intractable the problem, the farther off these outliers might be. Overall, the generated models were classified into categories like *"type I, VP constraints only,* 25% *constrainedness"* or *"type II, VP and C constraints,* 75% *constrainedness"*, for example. Figure 13.1 shows the plots that presents our results. The presented data is for simple SAT checks, where the SAT solver had to prove satisfiability. The Figure caption explains the presented data.

results for types I and II

The results shown in Figure 13.1 provide interesting insights. Overall, these results show that SAT checking scales well for well for models of type I as well as for models of type II (i.e., the required time for SAT checking increases only mildly super-linearly with the size of the model). Even for SPREBA models (type II) with 100% VP and C constraints (as defined above) and 930 decision items the required time for verifying the model's satisfiability is still at less than a second. We assume that these models are *far* more complex and larger than most real-world product lines. The examples we discussed in Sections 5.2, 6.2, 12.1.2, 12.2.2, and 12.3.1 were much smaller. The results also show that despite the significantly increased complexity of type II (SPREBA) models, from a human's point of view—weak dependencies can be hard to comprehend for humans because of their transitivity; and this aggravates when additional constraints are involved—, the required SAT solving time only increases by factors between about 1.5 (when no constraints are involved) to about up to 3 (when the maximal amount of constraints is involved). This is also interesting as it shows that the additional cross-cutting and weak dependencies in a SPREBA model (without additional constraints) only leads to an about 50% higher running time for a SAT check, compared to similar feature models, which are in a pure tree format.

results for type III

The results for the SAT checking performance on models of type III are further presented in Figure 13.2. Compared to models of type I and type II, Figure 13.2 shows a much more non-linear scalability. Models at the size of only about 200 decision items already require a longer time for a satisfiability check than the most complex category of type II, at a size of 930 decision items. When considering this heavily non-linear scalability, models of the size of around $1'000$ decision items will practically be unsolvable in any reasonable amount of time. The plot only shows the data for up to a size of 200 decision items because the model generator and the SAT solver's progress at this size was already very slow. Extending this plot with more data would have required several more days or weeks of running time.

easy vs. hard problems

When comparing the plots presented in Figure 13.1 to the one shown in Figure 13.2, we find a strong empirical evidence that models of the type I and II are actually well-behaved (i.e., they could be considered as 'easy' to analyze for a SAT solver). Models of the type III, on the other

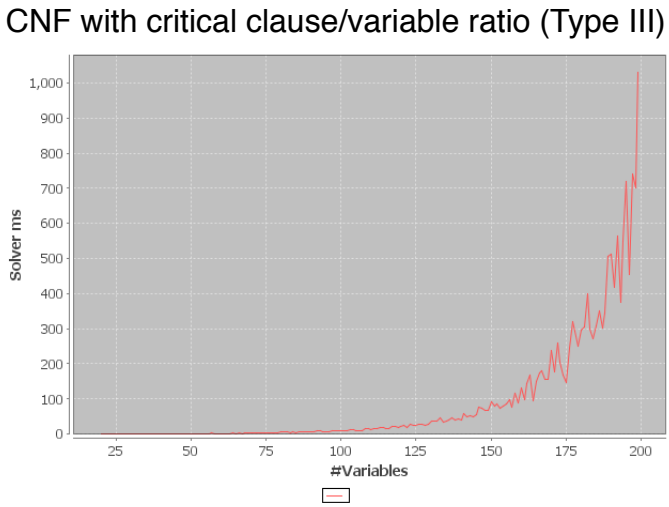


Figure 13.2: The aggregated SAT performance results for models of the types III, where the x-axis shows the number of decision items and the y-axis the time needed for SAT solving in milliseconds.

hand, are *not* well-behaved (i.e., they can be considered as 'hard' for a SAT solver, just as the theory has predicted for this particular clause-to-variables ratio).

conclusion—SPREBA scales

Overall, this empirical performance analysis has shown that SPREBA models—even with high amounts of cross-cutting, weak hierarchical dependencies and additional constraints—are actually *well-behaved* in practice, when automatically analyzed with a state-of-the-art SAT solver. This validates the feasibility of the SPREBA language in general (Chapters 5 and 6) and preliminarily also of SPREBA's SAT-based automated analysis solution (Chapter 9). A more detailed empirical analysis of SPREBA's more complex automated analysis operations (Section 9.4.2) should still be investigated in future research, however, as also suggested in Section 15.3.

CHAPTER 14

Case Study: Customization of ERP Systems

To demonstrate the real world feasibility and benefits of SPREBA, a case study within an ongoing software product line project has been performed. The general goal was to evaluate how well SPREBA (based on ADORA) and the state-of-the-art approach (i.e., feature modeling and UML) support software product line engineering activities in real-world projects. The focus was laid on evaluating strengths, weaknesses, and trade-offs between these two approaches. An industry cooperation with the Austrian company Inside AX GmbH—a partner company that distributes Microsoft's Dynamics AX—provided real-world data and access to engineers who customized this enterprise resource planning (ERP) system. The study was a collaborative effort between the Johannes Kepler University Linz, the University of Zurich, and Inside AX GmbH.

The study was conducted as a master's thesis [Schadauer, 2011]. The thesis was academically advised by Prof. Paul Grünbacher from the Johannes Kepler University Linz and by the author of this thesis. The modeling tools used were the ADORA tool (which implements the SPREBA approach) and Enterprise Architect from Sparx Systems Pty Ltd together with pure::variants from pure-systems GmbH for UML and feature modeling.

The results show a considerable potential and some clear benefits of SPREBA over the state-of-the-art approach. The particular advantages of using the SPREBA approach include a better understandability and more time savings. The UML and feature modeling approach was perceived an advantage where the stakeholders could leverage their existing knowledge of these concepts. For the other studied tasks no strong advantages or disadvantages for either approach were found. ADORA and SPREBA scored about equally well also in these other areas. The

mostly named reason for why SPREBA did not perform better was that the tool support was still too prototypical.

14.1 Case and Research Approach

creation of bids/orders subsystem

The Dynamics AX ERP system as a whole consists of an enormous amount of configuration possibilities (many thousands) and functional requirements. Thus, a particular part of Dynamics AX was selected as a representative subsystem for this study, namely the creation of bids/orders. This part includes features like *adding additional links to customers*, *adding additional values on a bill*, *adding discounts*, *adding alternative offered items*, *creating reports*, *selling items over Amazon's web service*, or *creating HTML text blocks for reports*, for example. Most of these features are already provided by Dynamics AX and can be realized with parameterization (i.e., by setting specific configuration keys). But some others are also realized as vertical solutions, which were additionally developed by Inside AX GmbH (the partner company for this study), to meet the very specific customer requirements.

Dynamics AX customization

Inside AX GmbH customizes and tailors Microsoft's Dynamics AX ERP system for particular industries and customer companies. Customization projects at InsideAX use the Microsoft Dynamics Sure Step Methodology[1] as a process model, where the requirements engineering and customization activities are mostly part of Sure Step's Diagnostic phase.
When Inside AX develops a customized Dynamics AX system for a specific customer company, a requirements analysis is performed first, which analyzes the business processes of the customer company. Then a so-called fit-gap analysis gets performed, which evaluates which requirements can be covered by the Dynamics AX system and for which requirements additional development efforts have to be performed. When an initial requirements specification is created, a first concept will be developed or a prototype will be realized. Using such a concept or prototype allows the analysis of hardware requirements and of additional commercial off-the-shelf (COTS) modules. Finally, a prototype of the envisioned ERP system will be presented to the customer company. Based on this prototype the customer company will decide whether it buys a full version of this prototype or whether it chooses a solution of a competitor.
The problem with the process at InsideAX was that the customization tasks for developing the first prototype were often very time-intensive and costly. Also, the initially agreed requirements specifications were often not accurate enough and needed to be re-negotiated later-on in development projects, which led to raised efforts. In ERP ecosystems the application of existing requirements modeling and software product line concepts is not straightforward, as also pointed out in [Nöbauer et al., 2012]. A related ongoing research project, called PL4X (Product Lines for AX) [Nöbauer et al., 2012], aims at addressing these issues. In Schadauer's work the focus was not on technical solution development, but rather on the requirements engineer-

[1] See http://www.microsoft.com/dynamics/support/implementation/success.aspx (checked on July 5, 2012).

ing phase [Schadauer, 2011]. In particular, the elicitation and the documentation of functional requirements and of commonality and variability was emphasized. Overall, a product line engineering approach is planned within Inside AX GmbH, with the general goal to achieve a rapid development of customized and working ERP system prototypes. Schadauer has scientifically evaluated the performance of the two approaches (i) SPREBA based on ADORA and (ii) feature modeling with UML in this context [Schadauer, 2011].

research approach

Methodologically, Schadauer performed action research and adhered to the principles of canonical action research [Davison et al., 2004]. From the viewpoint of this thesis, however, the work in [Schadauer, 2011] can also be viewed as a case study [Yin, 2002] because the author was not actively involved as a stakeholder, but only as an academic advisor. Research questions (RQs) were formulated to evaluate the performance of SPREBA and feature modeling in all product line engineering-relevant tasks. These tasks included requirements and variability elicitation and modeling, product derivation and product line maintenance and evolution tasks.

research questions

The following research questions were asked: 1) *How much effort is it to model the reference requirements model and the variability in every of the two approaches?* 2) *How well do the created models improve the understanding of variable features and the project as a whole?* 3) *How high is the required effort in every of the two approaches to derive valid products?* 4) *To what extent can the development time and effort for the development of features for a particular product be reduced?* And 5) *How big is the required effort for performing typical maintenance and evolution activities with every of the two product line models?*

GQM catalogue

Further, a goal question metric (GQM) [Basili et al., 1994] catalogue has been derived from these research questions, together with the academic supervisors and the company's stakeholders. Every research question asked got defined as a dedicated goal in this catalogue. For example, *'goal 1'* had the objective to *'create the requirements and variability modeling in UML and feature modeling and* ADORA *and SPREBA'*, the focus on the required *'effort'* and took the viewpoint of the *'modeler'*. All goals had the purpose to do a comparison between the two modeling approaches. To measure how well goal 1, for example, got satisfied by every approach, questions were derived, like, e.g., *'how high is the initially required effort for training?'*, *'how fast is it possible to model the requirements reference model?'*, or *'how laborious is it to model the variability of the requirements reference model?'*. Metrics were defined to allow a quantification of characteristics to systematically answer these questions. For example, the *'required time for familiarization and learning how to use the approach and tool'*, measured in *'working hours'*. Or the *'required effort to create the reference model and variability modeling'*, measured in both *'working hours'* and in the *'required number of single model edit operations'*. See [Schadauer, 2011] for a complete listing.

14.2 Results

Table 14.1: An overview of the case study's comparative evaluation results, taken from [Schadauer, 2011].

Goals (RQs)	Questions (abbreviated) *How big is the ... ?*	Metrics (abbreviated) *Measured by the ... .*	ADORA / SPREBA	UML / FM
RQ 1: Product line model creation	familiarization and learning effort	time needed	o	+
	reference model creation effort	time needed for and number of edit operations	o	o
	variability model creation effort	time needed for and number of edit operations	+	o
RQ 2: Understandability of product line models	required overall size of the modeling	number of model elements	+	o
	perceived understandability of the product line model	perceived understandability of dependencies between features	+	-
		time needed for deriving a product requirements specification	+	-
		perceived understandability of dependencies between requirements	+	-
		certainty about correct understanding of dependencies	+	-
	understandability of variability dependencies	understandability of dependencies between features	+	-
		time needed for determining the compatibility of features	o	+
	effort/duration for understanding the product line model	time needed to understand the modeling	-	+
		actually perceived understanding of the modeling (quality)	-	+
	effort for understanding the full impact of a variable feature	time needed to elaborate the feature impact	+	-
RQ 3: Product derivation	effort to derive a valid product	time to derive a desired valid product	o	+
	time / effort to check the correctness and validity of the derived product	time needed to review the derived product model	+	+
RQ 4: Time saving by using a product line model	ease of identifying components of a specific feature, based on the product line model	identifiability of relevant components via the model	+	-
		time savings for identifying relevant components via the model	+	-
	improvement of reusability of existing solutions, based on the product line model	time needed to identify equal requirements	+	+
RQ 5: Scenarios for product line maintenance and evolution tasks[1]	effort to realize the maintenance or evolution scenario	time needed to perform the scenario	o	o
		number of model edit operations to perform the scenario	o	o
	effort required to review the correctness of the scenario realization	time needed to review the changed model	o	o
	impact of the scenario realization on the model's size	increase/decrease in the number model elements	o	o

[1] **RQ 5** got evaluated for five scenarios: 1. adding a new feature, 2. changing or extending the specification of an existing feature, 3. adding a feature that overlaps with existing features, 4. turning a variable feature into a mandatory feature, and 5. changing the model of the commonality of the product line.

Table 14.1 presents an overview of Schadauer's empirical results. The table presents a very abstract description of the developed GQM catalogue (i.e., the columns *Goals*, *Questions*, and *Metrics*) and the results of [Schadauer, 2011]'s empirical evaluation for the two approaches evaluated within Inside AX GmbH (i.e., the columns ADORA / *SPREBA* and *UML / FM*). The results are only shown in a very abstract form, where *'+'* denotes a good performance, *'o'* a medium performance, and *'-'* a weak performance of the approach. These results got derived from a quantitative analysis (e.g., time needed for modeling and product line engineering tasks, amount of required operations, number of model elements required) and a qualitative analysis (e.g., stakeholders used the tools to perform specific requirements engineering and development tasks and were asked for feedback, impressions, and experience; and the modeler's own experience). Schadauer's overall results [Schadauer, 2011] for the five main research questions asked can briefly be summarized as follows.

product line model creation

Both approaches were evaluated to perform about equally well in the area of *product line model creation* (RQ 1). This slightly deviates from earlier results in [Zoller, 2010], where ADORA and SPREBA were found to better support the product line model creation and yielded an overall considerably smaller size. The reason for this deviation is that Schadauer's models were quite abstract, while Zoller's models (i.e., for the GoPhone case, recall Section 12.2.2) were rather detailed. SPREBA did not score higher because its tool implementation and view generation support was still cumbersome to use at some points and, hence, dampened the practical time advantages.

understandability

In terms of *understandability* (RQ 2) SPREBA models were perceived as significantly better by the company stakeholders because the traceability between requirements and variability was much more obvious. This lead to overall better results of SPREBA for understandability. On the other hand, those tasks, where the stakeholders needed to understand the semantics of the requirements modeling, yielded better results for the UML-based approach. The reason for this was that the stakeholders intuitively understood the UML modeling, but were troubled with the semantics of an ADORA model. A better previous knowledge of ADORA modeling (i.e., similar to the stakeholder's previous knowledge of the UML) would presumably have further improved the results of ADORA and SPREBA.

product derivation

In the area of *product derivation* (RQ 3) the focus was laid on the time needed to derive a product. Overall, slight benefits for the UML-based approach were found. SPREBA always generates a valid (partially) derived product (line) model whenever a new variability binding decision is taken (recall Chapter 8). This requires a short waiting time for feature weaving and the re-calculation of the newly required constraint propagation when a variability binding decision is taken. It therefore allows a better understanding of every decision's consequences, though, because it directly visualizes the impact of any taken decision. This integration of product generation and configuration, however, has not been perceived as success-critical in [Schadauer, 2011]. The general impression was that impact analysis is something that should be available when needed, but that is not crucially needed as the default mode of configuration.

On the other hand, how well every approach supports the understanding of a variability binding decision's impact was not appropriately covered with questions and metrics. Pure::variants only handles the constraint propagation during a product configuration and generates the product requirements model when the configuration is complete. This naturally allowed a faster derivation. The review of derived products was found to be well supported by both approaches, but this was only marginally investigated with the used GQM catalogue. As SPREBA's stepwise, incremental approach to product derivation generally required more time to reach a full configuration, Schadauer found the feature modeling and UML-based approach was slightly better in this category. A more qualitative comparison, however, may have yielded different results.

time savings

With regards to *time savings for an actual product development* (RQ 4) Schadauer found the SPREBA approach to perform very well, while the UML-based approach was found to be rather cumbersome. These results were not about deriving a product requirements model (RQ 3), but about efficiently identifying and utilizing solution space artifacts to realize a concrete product configuration. With ADORA and SPREBA it was particularly straightforward to identify the required ERP system customization activities for any selection or deselection of a variable feature. This information was directly specified in every feature aspect. In the UML-based approach, on the other hand, many separate diagrams needed to be studied to find out how a particular feature needs to be realized. This was found to be cumbersome and hardly beneficial (i.e., compared to only relying on ad hoc solutions or a textual documentation).

maintenance and evolution

Finally, for the various *maintenance and product line evolution* activities (RQ 5) both approaches were found to perform roughly equally well, with only slight differences between the five evaluated scenarios. The evaluated scenarios are mentioned in Table 14.1's *footnote 1*. In the case of SPREBA, these tasks are essentially well-supported, but the mere size of the model and the still prototypical state of the ADORA tool prevented a purely positive performance of the approach. In the UML-based approach, the tools were more reliable, but the scattering of the overall specification also hindered a purely positive performance of the approach. The ADORA tool could have scored considerably better with an industrial-strength tool implementation. But also the UML-based approach could benefit from additional, advanced tool support in this area, which may be similar to some of the concepts developed within the SPREBA approach.

threats to validity

This case study has yielded valuable insights on the actual performance of state-of-the-art requirements and variability modeling paradigms. However, there are several threats to validity [Wohlin et al., 2000] for this study, as also discussed in [Schadauer, 2011]. Most importantly, the results of this study are still of limited external validity because the studied case was not a classic software product line (i.e., where the company essentially owns the software product line). Instead, the studied case was the development of a product line approach in a partner company of the ERP system vendor, where the ERP system can be regarded a software ecosystem [Bosch and Bosch-Sijtsema, 2010] [Nöbauer et al., 2012]. For classic software product lines, where development is the primary concern, instead of configuration, tailoring, and enhance-

ment, we may find slightly diverging results (see [Zoller, 2010], for example). Thus, future empirical research will still be necessary to affirm and generalize these results. Other threats to validity are that there were only two company stakeholders involved, which yields only a low statistical significance (conclusion validity). One of these two stakeholders was also involved in both the elicitation of the model's data and its evaluation, which could have biased the outcome (internal validity). Both stakeholders were also already experienced with the UML and not with ADORA, which may have favored better results for the UML (construct validity). Eventually, the fact that the used UML and feature modeling tools were industrial-strength software tools and that the ADORA tool was only developed as a research prototype, that still contained bugs, may also have favored better results for the UML (construct validity).

overall results

In general, Schadauer's results showed that the SPREBA approach bears several considerable advantages to the existing state-of-the-art approach, when applied within a real-world project. These advantages could particularly be found for the *understandability* of the product line model and for *time savings* when using the product line model to develop the actual products. The study, however, has also found strengths of the state-of-the-art approach, which particularly were the use of widely known languages and notations and a lower visual complexity of the used graphical diagrams. With a more sophisticated tool implementation of SPREBA's view generation capabilities and with a better training in ADORA and SPREBA modeling these yet remaining drawbacks of ADORA and SPREBA could still be alleviated in the future, however. The SPREBA concepts, hence, overall bear a significant potential for improving the state of the art in requirements and variability modeling to support software product line engineering-related activities.

Part IV

Conclusions and Outlook

CHAPTER 15

Conclusion and Future Work

15.1 Summary and Contributions

addressed problem(s)

Today's widely known requirements modeling languages, with the UML as the most prominent one (Section 2.2), typically distribute a requirements model into multiple complementary diagrams of different types, to specify the different facets of a system (e.g., structure, behavior, user interaction, etc.). Since variability of a software system typically impacts multiple or all of these facets of a software system, almost all existing variability modeling techniques take an orthogonal and mapping-based approach to specify variability (Section 3.2). This inevitably leads to an information scattering of the specification of variable features—the detailed requirements and variability specification of any variable feature is scattered over many diagrams of different types (Figure 4.1). Consequently, this information scattering also leads to unnecessarily heightened efforts for 'specification and consistency maintenance' and 'impact comprehension of variability binding decisions', as highlighted in Chapter 4.

proposed solution / thesis

This thesis addressed this fundamental problem of information scattering of the variability specification and presented an *integrated* and *compositional* approach to model variability, supported by *view generation* to leverage the new approach's practical scalability (Chapter 5). Building on an integrated modeling language that uses only a single, integrated notation allows modeling and visualizing all facets of a variable feature's specification in one diagram. Using a compositional approach to modularize variable features also replaces the need for any

additional variability modeling notation and, hence, completely avoids information scattering. However, powerful abstraction and view generation techniques are necessary, to allow such an approach to scale also for larger models. ADORA is an example language and tool that provides all these prerequisites. On this basis a new, integrated approach to requirements and variability modeling is presented in this thesis (Part II).

realization of the approach

We developed a parsimonious language novel concept—*Boolean decision modeling*—to allow a fully-fledged *variability modeling* based on an integrated modeling language and a compositional approach. Chapter 6 presents these language extensions, where the ADORA language (Section 2.3) and ADORA's aspect-oriented modeling capabilities (Section 2.4.2) were used as a basis. They can be realized with any modeling language that satisfies the prerequisites as described in Chapter 5, however. Further, we developed various novel concepts for tool support for major product line engineering activities. We extended ADORA's aspect weaving capabilities to allow a stepwise and incremental product derivation (Chapters 7 and 8). We developed a sophisticated automated analysis solution to automatically resolve, analyze, and propagate the variability constraints whenever necessary (Chapters 8 and 9). We created a new solution for semi-automatic, incremental variability extraction (Chapter 10). Furthermore, we implemented a tool that realizes all these concepts and, hence, proves the technical feasibility of this new approach (Chapter 11). We called this approach SPREBA, which stands for Software Product line Requirements Engineering Based on Aspects [Glinz, 2008b]. SPREBA solves the addressed problems of state-of-the-art approaches, as follows.

no information scattering

The SPREBA approach fully avoids the scattering of a feature's specification over multiple separate diagrams. Figure 5.1 shows how this is realized by building on the principles of *integration* and *composition*. SPREBA allows modeling a variable feature and its detailed specification in a single aspect container (i.e., which is also called a *feature* when declared 'variable') that can further contain arbitrarily many nested aspects (i.e., also called 'parts of a feature') to provide a precise and accurate feature specification. This way the complete specification of a variable feature can be shown in a single view. All variable features and all variability constraints can be visualized in the same diagram as well. This solves the first and the most fundamental problem of *information scattering* (Section 4.1). However, advanced support for view generation (Sections 2.3.3 and 5.1) is crucial in this context to keep the model reasonably abstract and to avoid an information overflow to the user, which is required to ensure the approach's scalability.

efficient specification and maintenance

Using an integrated modeling language (i.e., one where the whole model can be shown with an integrated concrete syntax and visual notation) and a compositional approach to variability modeling allows new concepts for semi-automated variability modeling to be developed. These only require a manually provided selection of model elements, by a domain expert, and can completely automate the compositional variability specification (i.e., the creation of all required aspects, the creation of the required weaving semantics, and the extraction of all selected model elements). This reduces the necessary clerical and intellectual effort for compositional variability modeling substantially. Hitherto empirical results showed that the efforts for

variability-related activities could be reduced by factors between 2.8 and 4.8 already for small product lines and with only three products (see Chapter 12 and [Stoiber and Glinz, 2010a]). Larger product portfolios would yield even higher benefits, though. Moreover, a case study has shown that this approach also bears considerable benefits when compared to state-of-the-art industrial-strength tools in real-world projects, despite the still prototypical state of its tool implementation (Chapter 14). These empirical results show that the presented approach also solves the second problem of *high specification and consistency maintenance efforts* (Section 4.2), to a significant extent.

improved impact comprehension

Building on a fully integrated concrete syntax allows a more compact and straightforward visualization of a variability binding decision's impact. Calculating the complete set of constraint propagations (Section 9.4.2) together with weaving, removing, and/or re-visualizing particular variable features (or parts of features, Chapter 7) further allows to better comprehend the impact of any variability binding decision. The effects, both on the product's functional specification and on the selectability of other variable features (through variability constraints) can completely be visualized in a single, integrated view. While this may not always scale for large features in very detailed specifications, other ideas like so-called *impact views* [Stoiber et al., 2008, Chapter 4] can further be elaborated to allow a higher flexibility and scalability of such an integrated modeling approach. Hence, this solution also solves the third problem of *weak impact comprehension of variability binding decisions* (Section 4.3).

conclusion

Overall, this thesis has outlined the state of the art in requirements and variability modeling and highlighted considerable open problems that emerge from using a fragmented approach to requirements modeling (i.e., separate diagrams) and a mapping-based approach for variability modeling. To address and to solve these problems a new type of requirements and variability modeling approach has been developed, which builds on integrated requirements modeling and compositional variability modeling. This approach also supports crucial automated variability analysis mechanisms. The presented empirical validation has proven the technical and empirical feasibility of the approach and has also revealed considerable real-world benefits.

We believe that these presented paradigms will influence future requirements and variability modeling languages and approaches. These concepts could significantly change and improve the way engineers visualize, browse, and use product line requirements specifications. However, the actual usability of this approach stands and falls with the expressibility and readability of the used specification language and, in particular, with the performance and usability of the used view generation approach. Ideally, advanced tool support allows the generation of any view of interest at any abstraction level an engineer may desire and always guarantees a fully consistent and correct view on the model (hence, never shows a view that communicates the model wrongly). This requires a very human-friendly and stable implementation. Such an ideal implementation would allow navigating the model and changing perspectives as seamlessly, easily, and accurately as possible. When these conditions are provided, we believe the presented approach will outperform any state-of-the-art requirements and variability modeling solution.

15.2 Limitations

The SPREBA approach, as presented in this thesis, bears a considerable potential for improving state-of-the-art requirements and variability modeling solutions, but still carries several limitations, today. The most important ones are the following.

ADORA

So far, SPREBA has only been realized on the basis of ADORA, in the ADORA tool. While ADORA is in theory very expressible and suitable for all kinds of requirements specifications, it has failed to be adopted for industry projects within the last decade. Practitioners argue that the language is too formal and too laborious to use. It is typically more cost-efficient to build on just enough text-based requirements specification that is possibly augmented with diagrams. Because ADORA itself has not yet been adopted in industry it is unrealistic to expect that SPREBA on the basis of ADORA will be adopted in real-world projects any time soon. This is a strong limitation to the presented approach. However, we believe that future research on very lightweight modeling [Glinz, 2010b], which allows an easy documentation of requirements with only little training efforts, could possibly help to overcome this strong limitation in the future. We also hope that other modeling languages and approaches will adapt and exploit the SPREBA variability modeling paradigms as well.

abstraction and view generation

Advanced view generation concepts for integrated modeling have been developed since [Berner et al., 1998a] and were most recently refined in [Reinhard, 2010]. However, in practice, requirements modeling with ADORA still hardly scales better than UML modeling with state-of-the-art tools, for large models. The UML handles the visual complexity of large models quite well, by splitting the model into many complementary and hence smaller and simpler diagrams. Experience with the current version of the ADORA tool (e.g., in [Zoller, 2010] or [Schadauer, 2011]) has shown that there is still much room for improving the performance and usability of the tool. Especially when considering aspect weaving, maintaining the mental map of the model, and aiming for always well-arranged and easily human-readable diagrams [Kandrical, 2009] together, building optimal and highly efficient tooling concepts becomes a difficult problem that is still unsolved. Hence, the available tool implementation's usability and capabilities for the generation of arbitrary and human-friendly abstract views is still a limiting factor, today.

composition: expressibility

Most aspect-oriented modeling solutions that build on a join point model have a limited expressibility, which includes ADORA's AOM solution. Only limited combinations of model fragments can efficiently be extracted and modeled, but not arbitrary ones. Particularly, the aspect-oriented modeling of behavior chunks in ADORA has proved to be cumbersome [Zoller, 2010] [Jehle, 2010], as also discussed in Section 12.4. This still limits the compositional expressibility of variability in ADORA, today. Recent advances in aspect-oriented modeling, such as MATA [Whittle et al., 2009], for example, could help to solve this limitation and allow the modeling of *any* composition semantics that may be required. While MATA would allow more general model transformations, it may also yield additional complexity costs, however. In general, to-

day compositional approaches for fully integrated modeling languages, hence, are still either of limited expressibility or practicality.

SAT-based analysis performance

The experiments presented in Chapter 13 show that the SAT-based automated analysis of SPREBA models—which also include weak hierarchical dependencies and complex variability constraints—is well-behaved and scales quite well for any valid SPREBA model. However, this validation has only studied simple SAT checks of domain models. The presented algorithms for SAT-based automated analysis as presented in Chapter 9 (e.g., finding dead or mandatory decision items, constraint propagations, or automatically resolving conflicts with minimal change sets) typically require more than only simple SAT checks and may also include partial assignments. While we expect the SAT solving of SPREBA models with partial assignments to be even faster (i.e., the formula Φ can automatically be pruned by the solver in these cases) this has not yet been evaluated. Also, the actual performance of our solution to calculate minimal change sets, in order to resolve manually induced conflicts, is as yet unknown. This is still a limit to the approach's empirical validation as well.

tool support

SPREBA has been implemented with the ADORA tool, which has always been an evolutionary research prototype. Hence, the tool can naturally not compete with industrial-strength tools, which had much higher development budgets and typically do not implement functionalities as complex as the ADORA tool does (e.g., graphical feature weaving, view generation that preserves the mental map, or SAT-based automated analysis that includes cross-cutting and weak dependencies). The ADORA tool is a complete tool implementation of these concepts, though. However, the existing prototypical quality and usability of this tool implementation is still a significant limitation to the approach's real world applicability, today.

empirical validation

Once the above mentioned limitations are addressed, a more comprehensive empirical validation of the approach would be required. Some parts of the validation presented in Part III are still rather preliminary. For example, the performance evaluation presented in Chapter 13 only presented the results for simple SAT checks of the SPREBA domain model, as discussed above. Furthermore, the case study presented in Chapter 14 was performed in the context of a software ecosystem [Nöbauer et al., 2012], rather than a classic software product line. While our empirical evaluation nevertheless is a solid proof of concept of all the major contributions of this thesis, this empirical validity is still limited, to some extent. Improved tool support and further case studies could provide a significantly stronger empirical validation in the future.

15.3 Outlook on Future Research

Next to many smaller open issues to optimize, we suggest four major research directions for future research, as follows. Most of them are major research endeavors that go beyond the viable scope of this thesis.

views and layouts

ADORA's existing graphical aspect weaving solution (see Section 2.4.2, [Meier, 2009], and [Kandrical, 2009]) is still an unsolved problem in the search for an optimal and human-friendly solution. While we still lack an ideal and stable solution for this purpose, we identified three potential solution ideas, as follows. First, the diagram could be layouted efficiently and human-friendly by re-arranging larger parts of the model or the model as a whole with strict layouting guidelines, every time an aspect is woven. This would allow human-friendly and consistent layouts, but comes at the cost of losing the mental map of the model, unfortunately—Kandrical has explored this solution to some extent [Kandrical, 2009]. Second, the aspects could always be woven in a way that preserves the mental map of the model as far as possible. This maintains the mental map, but leads to sub-optimal and often inconsistent layouts that generate too much white space and are often not too human-friendly. Among these two solution strategies we think that in most cases a trade-off will be ideal. When performing many feature weaving operations subsequently, however, then finding an optimal (or close to optimal) solution for any model becomes tricky to be automated in a tool (i.e., when dealing with large models). A third solution, that could possibly always generate ideal graphic layouts, might be that *all* aspects are already woven during domain model maintenance activities and that the overall graphic layout is fully manually optimized by human engineers. Then, all aspects could be re-extracted, but the woven model elements would still be kept as infinitesimally small and hidden elements in the fully woven model—similar to Reinhard's view generation solution [Reinhard, 2010]. This way, weaving an aspect would only mean re-visualizing the already woven and positioned model elements, which already yield a good overall layout for at least one human engineer (the original layouter). This solution requires large amounts of manual layouting work, however, and may therefore also be suboptimal.

The goal is that a graphic aspect weaving always fully automatically finds a good-enough trade-off that is very close to an optimal visual layout and preserves the mental map. This is difficult because—especially in hierarchical diagrams—changes may have an impact on several layers of the model. Furthermore, Reinecke, for example, has already shown that preferences of graphic layouts differ even by the engineer's cultural background (at least in graphical user interfaces) [Reinecke, 2010]. We hope that future research in this area will develop novel and powerful algorithms and concepts that will allow tool support to automatically generate a well-arranged model layout at any desired level of abstraction and in any desired partial or full variability configuration. The above mentioned strategies could be a step towards that goal, but may be difficult to realize in a reliable and stable fashion.

SAT-based analysis

The SAT-based automated analysis presented in Chapter 9 is a novel, comprehensive solution for the automated analysis of SPREBA models and Boolean variability models in general. While our descriptions are already solid, we think that this solution can still be improved with a deeper theoretical analysis, however. For example, Liffiton and Sakallah described how minimal unsatisfiable cores can be found for constraint satisfaction problems [Liffiton and Sakallah, 2008]. A minimal unsatisfiable core is the minimal set of variables to which the cause of unsatisfiability can be reduced (i.e., changing any other variable will not resolve the unsatisfiability).

To resolve user-induced conflicts, such a solution that relies on minimal unsatisfiable cores could be used to improve Algorithm 6, for example. An improved solution in this regard could be similar to how the Alloy analyzer generates counter examples, see [Shlyakhter et al., 2003]. Further, synergies between the solution in this thesis and the one presented in [Xiong et al., 2011] could also be explored. Moreover, it should be investigated whether SAT solving is really optimal or whether a more efficient basis for the operations presented in Section 9.4 can be found (SAT seems to be highly optimized for these kinds of tasks, though). Future research should perform a deeper theoretical analysis in this regard and also present more exhaustive empirical data for all the diverse automated variability analysis operations that are necessary to optimally support the software product line requirements modeling process.

visualizing existing models with SPREBA

Manually creating new requirements and variability models in the ADORA and SPREBA notation, while UML and feature models may already be in use for many years, is probably unrealistic for an application of SPREBA in industry. When these UML and feature models already exist, however, it could be an option to directly exploit SPREBA's benefits on their basis. For example, one could develop a tool to automatically generate ADORA and SPREBA models from existing UML and feature models. Alternatively, one could directly build on existing models and develop an additional concrete syntax and visual notation [Kleppe, 2008] that satisfies all SPREBA prerequisites as listed in Chapter 5. This would technically enable all benefits of the SPREBA approach also for any existing models. From a practical point of view, such future research could make the likelihood of an industry adoption of the SPREBA concepts much more probable. This would essentially require a new concrete syntax and visual notation for the UML, which is integrated as the one of ADORA is, recall Sections 2.2 and 2.3. Such an application would also provide invaluable empirical data and experience that could greatly help to further improve SPREBA.
Evolving SPREBA into an advanced integrated visualization approach that can also deal with existing models written in state-of-the-art languages (e.g., AHEAD, UML, and feature modeling, OVM, CVL or others) could have many benefits. For example, all existing tools for model-driven engineering and for any other purpose could still be used and all specifications would still adhere to today's industry standards. However, creating an ADORA-like concrete syntax for the UML's abstract syntax, for example, is most probably a highly challenging and large endeavor. Nevertheless, we think that developing such a new integrated concrete syntax and visual notation for existing languages would be very worthwhile, along with developing robust tool support. This could allow exploiting the benefits of SPREBA with only little manual effort for already existing models in industry.

collaboration support

Ultimately, another idea and area of future research, that may be quite fruitful, is dedicated tool support for a *collaborative* commonality and variability elicitation, modeling, and evolution. Some initial work in this area has been proposed in [Stoiber and Glinz, 2009] and [Glinz, 2010a], for example. The main idea was to merge ADORA and SPREBA modeling with an approach that is similar to the EasyWinWin approach [Boehm et al., 2001]. This thesis has

essentially always taken the assumption that only a single engineer or a co-located team of engineers performs all the product line engineering-related tasks. The presented tool support ideally supports such a situation. This assumption is unrealistic, however, as more and more real-world software projects involve key stakeholders in various geographical locations. A collaborative elicitation and negotiation of commonality and variability for an emerging software product line would be much more appropriate, though. Such a solution should be based on a single, central SPREBA model that everybody can view, criticize, and suggest improvements and evolutionary changes for. These suggestions could then centrally be collected, negotiated, and eventually be agreed upon in a systematic and organized manner. This would require flexible and user-friendly tool support that allows many or all stakeholders to browse the latest version of the model and to suggest any changes or new specifications based on EasyWinWin's solution to negotiate ideas and win conditions and to find mutual agreements [Boehm et al., 2001]. Such mutually agreed parts of the model could then be regarded as the actual requirements [Boehm et al., 1994]. We have performed preliminary experiments that have shown the general feasibility of this idea. Hence, we believe that such an evolution of the SPREBA approach would yield considerable improvements. It would allow a systematic collaboration between all project stakeholders, to find *win-win* agreements for all commonality requirements and for all variable features and their specification. In particular, this would most likely be the case for very early requirements engineering and product management phases. We expect that many new ideas would be raised during such collaborative elicitation processes, compared to a classic software product line requirements engineering process. Overall, we think such tool support for collaborative requirements and variability elicitation and specification could considerably improve both the quality and the effectiveness when applying SPREBA in the real world.

Bibliography

Apel, S., Batory, D., and Rosenmüller, M. (2006). On the structure of crosscutting concerns: using aspects or collaborations. In *1st Workshop on Aspect-Oriented Product Line Engineering at GPCE'06*.

Apel, S. and Kästner, C. (2009). An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84.

Apel, S., Kästner, C., and Lengauer, C. (2009). FeatureHouse: Language-independent, automated software composition. In *31st International Conference on Software Engineering*, pages 221–231.

Arango, G. F. (1988). *Domain engineering for software reuse.* PhD thesis, University of California at Irvine.

Asikainen, T., Männistö, T., and Soininen, T. (2004a). Using a configurator for modelling and configuring software product lines based on feature models. In *Workshop on Software Variability Management for Product Derivation at SPLC'04*, pages 24–35.

Asikainen, T., Soininen, T., and Männistö, T. (2004b). A Koala-based approach for modelling and deploying configurable software product families. In *5th International Workshop on Software Product-Family Engineering*, pages 225–249.

Atkinson, C. (2002). *Component-based product line engineering with UML.* Addison-Wesley.

Atkinson, C. and Stoll, D. (2008). Orthographic modeling environment. In *11th International Conference on Fundamental Approaches to Software Engineering*, pages 93–96.

Atkinson, C., Stoll, D., and Bostan, P. (2010). Orthographic software modeling: a practical approach to view-based development. In *Evaluation of Novel Approaches to Software Engineering*, pages 206–219.

Bae, J. H. and Chae, H. S. (2008). UMLSlicer: a tool for modularizing the UML metamodel using slicing. In *8th IEEE International Conference on Computer and Information Technology*, pages 772–777.

Balzer, R. (1991). Tolerating inconsistency. In *13th International Conference on Software Engineering*, pages 158–165.

Barstow, D. R. (1985). Domain-specific automatic programming. *IEEE Trans. Softw. Eng.*, 11:1321–1336.

Basili, V., Caldiera, G., and Rombach, H. (1994). The goal question metric approach. *Encyclopedia of Software Engineering*, 1:528–532.

Batory, D., Barnett, J., Roy, J., Twichell, B., and Garza, J. (1989). Construction of file management systems from software components. In *13th Annual International Computer Software and Applications Conference*, pages 358 –364.

Batory, D., Benavides, D., and Ruiz-Cortes, A. (2006). Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49:45–47.

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371.

Batory, D. S. (2005). Feature models, grammars, and propositional formulas. In *9th International Software Product Line Conference*, pages 7–20.

Benavides, D., Martín-Arroyo, P. T., and Cortés, A. R. (2005). Automated reasoning on feature models. In *17th International Conference on Advanced Information Systems Engineering*, pages 491–503.

Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6):615–636.

Berner, S. (2002). *Modellvisualisierung für die Spezifikationssprache ADORA [Model visualization for the specification language ADORA (in German)]*. PhD thesis, Department of Informatics, University of Zurich.

Berner, S., Joos, S., Glinz, M., and Arnold, M. (1998a). A visualization concept for hierarchical object models. In *13th IEEE International Conference on Automated Software Engineering*, pages 225–228.

Berner, S., Joos, S., Glinz, M., and Arnold, M. (1998b). Visualizing ADORA models. Technical report, TR-98-09, Department of Informatics, University of Zurich.

Berner, S., Schett, N., Xia, Y., and Glinz, M. (1999). An experimental validation of the ADORA language. Technical report, ifi-1999.07, Department of Informatics, University of Zurich.

Beuche, D., Papajewski, H., and Schröder-Preikschat, W. (2004). Variability management with feature models. *Sci. Comput. Program.*, 53(3):333–352.

Boehm, B. (1976). Software engineering. *IEEE Transactions on Computers*, C-25(12):1226 –1241.

Boehm, B. and Basili, V. R. (2001). Software defect reduction top 10 list. *Computer*, 34(1):135–137.

Boehm, B., Bose, P., Horowitz, E., and Lee, M.-J. (1994). Software requirements as negotiated win conditions. In *1st International Conference on Requirements Engineering*, pages 74 –83.

Boehm, B., Grunbacher, P., and Briggs, R. (2001). Developing groupware for requirements negotiation: lessons learned. *IEEE Software*, 18(3):46 –55.

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice Hall, 1st edition.

Bontemps, Y., Heymans, P., Schobbens, P.-Y., and Trigaux, J.-C. (2005). Generic semantics of feature diagrams variants. In *Feature Interactions in Telecommunications and Software Systems VIII*, pages 58–77.

Bosch, J. and Bosch-Sijtsema, P. (2010). From integration to composition: on the impact of software product lines, global development and ecosystems. *J. Syst. Softw.*, 83:67–76.

Botterweck, G., O'Brien, L., and Thiel, S. (2007). Model-driven derivation of product architectures. In *22nd International Conference on Automated Software Engineering*, pages 469–472.

Brownsword, L. and Clements, P. (1996). A case study in successful product line development. Technical report, cmu/sei-96-tr-016, Software Engineering Institute, Carnegie Mellon University.

Bühne, S., Lauenroth, K., and Pohl, K. (2004). Why is it not sufficient to model requirements variability with feature models? In *Automotive Requirements Engineering Workshop at RE'04*, pages 5–12.

Campbell, G., Faulk, S., and Weiss, D. (1990). Introduction to synthesis. Technical report, intro_synthesis_process-90019-n, Software Productivity Consortium.

Chen, L., Ali Babar, M., and Ali, N. (2009). Variability management in software product lines: a systematic review. In *13th International Software Product Line Conference*, pages 81–90.

Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M., Bakker, J., Tekinerdogan, B., Clarke, S., and Jackson, A. (2005). Survey of aspect-oriented analysis and design approaches. Technical report, aosd-europe-ulanc-9, AOSD-Europe.

Chomsky, N. (1965). *Aspects of the theory of syntax*. MIT Press.

Clarke, S. and Baniassad, E. (2005). *Aspect-oriented analysis and design*. Addison-Wesley.

Classen, A., Heymans, P., Schobbens, P.-Y., and Legay, A. (2011). Symbolic model checking of software product lines. In *33rd International Conference on Software Engineering*, pages 321–330.

Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., and Raskin, J.-F. (2010). Model checking lots of systems: efficient verification of temporal properties in software product lines. In *32nd International Conference on Software Engineering*, pages 335–344.

Clements, P. C. and Northrop, L. (2001). *Software product lines: practices and patterns*. Addison-Wesley.

Colyer, A. and Clement, A. (2004). Large-scale AOSD for middleware. In *3rd International Conference on Aspect-oriented Software Development*, pages 56–65.

Colyer, A., Rashid, A., and Blair, G. (2004). On the separation of concerns in program families. Technical report, COMP-001-2004, Computing Department, Lancaster University.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *3rd ACM Symposium on Theory of Computing*, pages 151–158.

Cottenier, T. (2006). The Motorola WEAVR: model weaving in a large industrial context. In *5th International Conference on Aspect-Oriented Software Development, Industry Track*, 10 pages.

Cramer, C. (2007). Verwendung und Prüfung von Integritätsbedingungen in der Modellierungssprache ADORA [Use and checking of integrity conditions in the modeling language ADORA (in German)]. Master's thesis, Department of Informatics, University of Zurich.

Czarnecki, K. and Antkiewicz, M. (2005). Mapping features to models: a template approach based on superimposed variants. In *4th International Conference on Generative Programming and Component Engineering*, pages 422–437.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming – methods, tools and applications*. Addison-Wesley.

Czarnecki, K., Gruenbacher, P., Rabiser, R., Schmid, K., and Wasowski, A. (2012). Cool features and tough decisions: a comparison of variability modeling approaches. In *6th International Workshop on Variability Modelling of Software-intensive Systems*, pages 173–182.

Czarnecki, K., Helsen, S., and Eisenecker, U. (2005a). Staged configuration through specialization and multi-level configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143-169.

Czarnecki, K., Helsen, S., and Eisenecker, U. W. (2005b). Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29.

Czarnecki, K. and Pietroszek, K. (2006). Verifying feature-based model templates against well-formedness OCL constraints. In *5th international conference on Generative programming and component engineering*, pages 211–220.

Czarnecki, K., She, S., and Wasowski, A. (2008). Sample spaces and feature models: there and back again. In *12th International Software Product Line Conference*, pages 22 –31.

Czarnecki, K. and Wasowski, A. (2007). Feature diagrams and logics: there and back again. In *11th International Software Product Line Conference*, pages 23–34.

Davis, A. M. (2005). *Just enough requirements management: where software development meets marketing*, Dorset House Publishing.

Davison, R. M., Martinsons, M. G., and Kock, N. (2004). Principles of canonical action research. *Inf. Syst. J.*, 14(1): 65–86.

Dhungana, D., Heymans, P., and Rabiser, R. (2010). A formal semantics for decision-oriented variability modeling with DOPLER. In *4th International Workshop on Variability Modelling of Software-Intensive Systems*, pages 29–35.

Dhungana, D., Seichter, D., Botterweck, G., Rabiser, R., Gruenbacher, P., Benavides, D., and Galindo, J. A. (2011). Configuration of multi product lines by bridging heterogeneous variability modeling approaches. In *15th International Software Product Line Conference*, pages 120–129.

Diaz, D. and Codognet, P. (2001). Design and implementation of the GNU prolog system. *Journal of Functional and Logic Programming*, 2001(6).

Dijkstra, E. W. (1972). Notes on structured programming. In *Structured programming*. Academic Press Ltd., pages 1–82.

Dijkstra, E. W. (1976). Executional abstraction. In *A Discipline of Programming*, Prentice-Hall.

Dutoit, A. H., McCall, R., Mistrik, I., and Paech, B. (2006). *Rationale management in software engineering*. Springer.

Eades, P., Lai, W., Misue, K., and Sugiyama, K. (1991). Preserving the mental map of a diagram. *Proceedings of Compugraphics*, 91: 24–33.

Egyed, A. (2006). Instant consistency checking for the UML. In *28th International Conference on Software Engineering*, pages 381–390.

Egyed, A. (2007). Fixing inconsistencies in UML design models. In *29th International Conference on Software Engineering*, pages 292–301.

Egyed, A., Letier, E., and Finkelstein, A. (2008). Generating and evaluating choices for fixing inconsistencies in UML design models. In *23rd International Conference on Automated Software Engineering*, pages 99–108.

Endres, A. (1975). An analysis of errors and their causes in system programs. *IEEE Trans. Software Eng.* 1(2):140-149.

Fantechi, A. and Gnesi, S. (2007). A behavioural model for product families. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 521–524.

Fantechi, A. and Gnesi, S. (2008). Formal modeling for product families engineering. In *12th International Software Product Line Conference*, pages 193–202.

Favre, J.-m. (2004). Towards a basic theory to model model driven engineering. In *Workshop on Software Model Engineering at UML'04*, 8 pages.

Felfernig, A., Friedrich, G., Jannach, D., and Zanker, M. (2001). Intelligent support for interactive configuration of mass-customized products. In *Proceedings of the 14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Engineering of Intelligent Systems*, pages 746–756.

France, R., Ray, I., Georg, G., and Ghosh, S. (2004). Aspect-oriented approach to early design modelling. *IEE Proceedings Software*, 151(4):173–185.

Fricker, S. and Schumacher, S. (2011). Variability-based release planning. In *2nd International Conference on Software Business*, pages 181–186.

Fricker, S. and Stoiber, R. (2008). Relating product line context to requirements engineering processes using design rationale. In *Software Engineering 2008 (Workshops)*, pages 240–251.

Garg, A., Critchlow, M., Chen, P., Van der Westhuizen, C., and van der Hoek, A. (2003). An environment for managing evolving product line architectures. In *19th International Conference on Software Maintenance*, pages 358–367.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002). *Fundamentals of software engineering*. Prentice Hall, 2nd edition.

Glinz, M. (1995). An integrated formal model of scenarios based on statecharts. In *5th European Software Engineering Conference*, pages 254–271.

Glinz, M. (2000). Problems and deficiencies of UML as a requirements specification language. In *10th International Workshop on Software Specification and Design*, pages 11–22.

Glinz, M. (2005). Informatics IIa: modeling course notes. Department of Informatics, Univerity of Zurich.

Glinz, M. (2006). Requirements engineering I course notes. Department of Informatics, Univerity of Zurich.

Glinz, M. (2007). On non-functional requirements. In *15th International Requirements Engineering Conference*, pages 21–26.

Glinz, M. (2008a). A risk-based, value-oriented approach to quality requirements. *IEEE Software*, 25(2):34–41.

Glinz, M. (2008b). SPREBA—software product line requirements engineering based on aspects. Project proposal, Swiss National Science Foundation (SNSF), grant nr. 200021-121904.

Glinz, M. (2010a). SPREBA—software product line requirements engineering based on aspects. Project continuation proposal. Swiss National Science Foundation (SNSF), grant nr. 200021-132752.

Glinz, M. (2010b). Very lightweight requirements modeling. In *18th International Conference on Requirements Engineering*, pages 385–386.

Glinz, M., Berner, S., and Joos, S. (2002). Object-oriented modeling with ADORA. *Inf. Syst.*, 27(6):425–444.

Gomaa, H. (2005). *Designing software product lines with UML: from use cases to pattern-based software architectures*. Addison-Wesley.

Groher, I. and Voelter, M. (2007). Xweave: models and aspects in concert. In *10th International Workshop on Aspect-oriented Modeling*, pages 35–40.

Groher, I. and Voelter, M. (2009). Aspect-oriented model-driven software product line engineering. *Transactions on AOSD VI*, 5560:111–152.

Grossman, M., Aronson, J. E., and McCarthy, R. V. (2005). Does UML make the grade? Insights from the software development community. *Inf. Softw. Technol.*, 47:383–397.

Habr, J. (2008). Management von Designentscheiden mit ADORA [Management of design decisions with ADORA (in German)]. Bachelor's thesis, Department of Informatics, University of Zurich.

Hall, T., Beecham, S., and Rainer, A. (2002). Requirements problems in twelve software companies: An empirical analysis. *IEE Proceedings Software*, 149(5):153–160.

Hallsteinsen, S., Hinchey, M., Park, S., and Schmid, K. (2008). Dynamic software product lines. *Computer*, 41(4):93–95.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Harel, D. and Rumpe, B. (2004). Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72.

Haugen, Ø. and et al. (2010). Common variability language (CVL). *OMG Initial Submission*.

Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G. K., and Svendsen, A. (2008). Adding standardized variability to domain specific languages. In *12th International Software Product Line Conference*, pages 139–148.

Heidenreich, F., Henriksson, J., Johannes, J., and Zschaler, S. (2009). On language-independent model modularisation. *Transactions on AOSD VI*, 5560:39–82.

Heidenreich, F., Kopcsek, J., and Wende, C. (2008). Featuremapper: mapping features to models. In *Companion of the 30th International Conference on Software Engineering*, pages 943–944.

Heidenreich, F., Sánchez, P., Santos, J. a., Zschaler, S., Alférez, M., Araújo, J. a., Fuentes, L., Kulesza, U., Moreira, A., and Rashid, A. (2010). Relating feature models to other models of a software product line: a comparative study of Featuremapper and VML. *Transactions on AOSD VII*, pages 69–114.

Hendrickson, S. A. and van der Hoek, A. (2007). Modeling product line architectures through change sets and relationships. In *29th International Conference on Software Engineering*, pages 189–198.

Hubaux, A., Xiong, Y., and Czarnecki, K. (2012). A user survey of configuration challenges in Linux and eCos. In *6th International Workshop on Variability Modeling of Software-Intensive Systems*, pages 149–155.

IEEE (1998). IEEE recommended practice for software requirements specifications. Technical report, IEEE Standard 830-1998.

Jackson, D. (2006). *Software abstractions: logic, language, and analysis*. The MIT Press.

Jacobson, I. and Ng, P.-W. (2004). *Aspect-oriented software development with use cases*. Addison-Wesley.

Janota, M. (2008). Do SAT solvers make good configurators? In *1st Workshop on Analyses of Software Product Lines*, pages 191–195.

Janota, M. (2010). *SAT solving in interactive configuration*. PhD thesis, University College Dublin.

Janota, M., Kiniry, J., and Botterweck, G. (2008). Formal methods in software product lines: concepts, survey, and guidelines. Technical report, Lero-TR-SPL-2008-02, Lero.

Jayaraman, P., Whittle, J., Elkhodary, A., and Gomaa, H. (2007). Model composition in product lines and feature interaction detection using critical pair analysis. In *10th International Conference on Model Driven Engineering Languages and Systems*, pages 151–165.

Jeanneret, C., Glinz, M., and Baudry, B. (2011). Estimating footprints of model operations. In *33rd International Conference on Software Engineering*, pages 601–610.

Jehle, M. (2010). Feature unweaving: semi-automated aspect extraction in product line requirements engineering. Master's thesis, Department of Informatics, University of Zurich.

Joos, S. (2000). ADORA-*L* – *Eine Modellierungssprache zur Spezifikation von Software-Anforderungen [*ADORA-*L* – *a modeling language for the specification of software requirments (in German)]*. PhD thesis, Department of Informatics, University of Zurich.

Joos, S., Berner, S., and Glinz, M. (1997). Hierarchische Zerlegung in objektorientierten Spezifikationsmethoden [Hierarchic decomposition of object-oriented specification methods (in German)]. *Softwaretechnik-Trends*, 1(17):29–37.

Kandrical, A. (2009). Graphical weaving of aspects in product line requirements engineering. Master's thesis, Department of Informatics, University of Zurich.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU/SEI-90-TR-21, Carnegie Mellon University.

Kästner, C. (2010). *Virtual separation of concerns: toward preprocessors 2.0*. PhD thesis, University of Magdeburg.

Kasunic, M. (1992). Synthesis: a reuse-based software development methodology, process guide, version 1.0. Technical report, Software Productivity Consortium Services Corporation.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, pages 327–353.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming*, pages 220–242.

Kienzle, J., Al Abed, W., and Klein, J. (2009). Aspect-oriented multi-view modeling. In *8th International Conference on Aspect-oriented Software Development*, pages 87–98.

Kleppe, A. (2008). *Software language engineering: creating domain-specific languages using metamodels*. Addison-Wesley, 1st edition.

Kotonya, G. and Sommerville, I. (1998). *Requirements engineering processes and techniques*. Wiley, 1st edition.

Krueger, C. (2006). New methods in software product line development. In *10th International on Software Product Line Conference*, pages 95–102.

Krueger, C. W. (2002). Easing the transition to software mass customization. In *4th International Workshop on Software Product-Family Engineering*, pages 282–293.

Krueger, C. W. (2012). White papers from BigLever Software: Gears, http://www.biglever.com/learn/whitepapers.html (checked on July 5, 2012).

Lamsweerde, A. V. (2001). Goal-oriented requirements engineering: a guided tour. In *5th International Symposium on Requirements Engineering*, pages 249–262.

Lange, C. F. J. and Chaudron, M. R. V. (2006). Effects of defects in UML models: an experimental investigation. In *28th International Conference on Software Engineering*, pages 401–411.

Lauenroth, K. and Pohl, K. (2007). Towards automated consistency checks of product line requirements specifications. In *22nd International Conference on Automated Software Engineering*, pages 373–376.

Lauenroth, K. and Pohl, K. (2008). Dynamic consistency checking of domain requirements in product line engineering. In *16th International Requirements Engineering Conference*, pages 193–202.

Lauenroth, K., Pohl, K., and Toehning, S. (2009). Model checking of domain artifacts in product line engineering. In *24th International Conference on Automated Software Engineering*, pages 269–280.

Lee, Y.-Y., Lin, C.-C., and Yen, H.-C. (2006). Mental map preserving graph drawing using simulated annealing. In *Asia-Pacific Symposium on Information Visualisation*, pages 179–188.

Liffiton, M. H. and Sakallah, K. A. (2008). Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reason.*, 40:1–33.

Linden, F. J. v. d., Schmid, K., and Rommes, E. (2007). *Software product lines in action: the best industrial practice in product line engineering*. Springer.

Lisboa, L. B., Garcia, V. C., Lucrédio, D., de Almeida, E. S., de Lemos Meira, S. R., and de Mattos Fortes, R. P. (2010). A systematic review of domain analysis tools. *Inf. Softw. Technol.*, 52:1–13.

Ludewig, J. (2003). Models in software engineering – an introduction. *Software and Systems Modeling*, 2:5–14.

Mannion, M. (2002). Using first-order logic for product line model validation. In *2nd Software Product Line Conference*, pages 176–187.

Mazo, R., Grünbacher, P., Heider, W., Rabiser, R., Salinesi, C., and Diaz, D. (2011). Using constraint programming to verify DOPLER variability models. In *5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 97–103.

McCluskey, E. J. (1956). Minimization of boolean functions. *Bell System Technology Journal*, 35(5):1417–1444.

Meier, S. (2009). *Aspect-oriented requirements modeling*. PhD thesis, Department of Informatics, University of Zurich.

Meier, S., Reinhard, T., Seybold, C., and Glinz, M. (2006). Aspect-oriented modeling with integrated object models. In *Modellierung*, pages 129–144.

Meier, S., Reinhard, T., Stoiber, R., and Glinz, M. (2007). Modeling and evolving crosscutting concerns in adora. In *Workshop on Aspect-Oriented Requirements Engineering and Architecture Design at ICSE '07*.

Mendonca, M., Wąsowski, A., and Czarnecki, K. (2009). SAT-based analysis of feature models is easy. In *13th International Software Product Line Conference*, pages 231–240.

Metzger, A. and Pohl, K. (2007). Variability management in software product line engineering. In *Companion to the 29th International Conference on Software Engineering*, pages 186–187.

Moody, D. and Hillegersberg, J. (2009). Evaluating the visual syntax of UML: an analysis of the cognitive effectiveness of the UML family of diagrams. *Software Language Engineering*, Springer, pages 16–34.

Moody, D. L. (2009). The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35:756–779.

Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., and Jézéquel, J.-M. (2009). Weaving variability into domain metamodels. In *12th International Conference on Model Driven Engineering Languages and Systems*, pages 690–705.

Muthig, D., John, I., Anastasopoulos, M., Forster, T., Dörr, J., and Schmid, K. (2004). GoPhone – a software product line in the mobile phone domain. Technical report, IESE No. 025.04/E, Fraunhofer IESE.

Neighbors, J. M. (1980). *Software construction using components*. PhD thesis, University of California at Irvine.

Nöbauer, M., Seyff, N., Dhungana, D., and Stoiber, R. (2012). Managing variability of ERP ecosystems: research issues and solution ideas from Microsoft Dynamics AX. In *6th International Workshop on Variability Modeling of Software-intensive Systems*, pages 21–26.

Nöhrer, A. and Egyed, A. (2010). Conflict resolution strategies during product configuration. In *4th International Workshop on Variability Modelling of Software-intensive Systems*, pages 107–114.

Object Management Group (2010a). UML 2.3 infrastructure, URL: http://www.omg.org/spec/UML/2.3 (checked on July 5, 2012).

Object Management Group (2010b). UML 2.3 superstructure, URL: http://www.omg.org/spec/UML/2.3 (checked on July 5, 2012).

O'Callaghan, B., O'Sullivan, B., and Freuder, E. (2005). Generating corrective explanations for interactive constraint satisfaction. In *Principles and Practice of Constraint Programming*, Springer, pages 445–459.

Padmanabhan, P. (2002). A requirements engineering tool for product families. Master's thesis, Iowa State University.

Padmanabhan, P. and Lutz, R. R. (2005). Tool-supported verification of product line requirements. *Automated Software Engineering*, 12(4):447–465.

Papadopoulos, A. and O'Sullivan, B. (2008). Relaxations for compiled over-constrained problems. In *Principles and Practice of Constraint Programming*, pages 433–447.

Parnas, D. (1976). On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9.

Passos, L., Novakovic, M., Xiong, Y., Berger, T., Czarnecki, K., and Wąsowski, A. (2011). A study of non-boolean constraints in variability models of an embedded operating system. In *3rd Workshop on Feature-Oriented Software Development at SPLC'11*, pages 1–8.

Pohl, K. (2010). *Requirements engineering – fundamentals, principles, and techniques*. Springer.

Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software product line engineering: foundations, principles and techniques*. Springer.

Popovici, A., Gross, T., and Alonso, G. (2002). Dynamic weaving for aspect-oriented programming. In *1st International Conference on Aspect-Oriented Software Development*, pages 141–147.

Prehofer, C. (1997). Feature-oriented programming: a fresh look at objects. In *European Conference on Object-Oriented Programming*, pages 419–443.

Prieto-Diaz, R. (1988). Domain analysis for reusability. In *Software Reuse: Emerging Technology*, IEEE CS Press, pages 347–353.

Reinecke, K. (2010). *Culturally adaptive user interfaces*. PhD thesis, Department of Informatics, University of Zurich.

Reinhard, T. (2010). *Complexity management in graphical models*. PhD thesis, Department of Informatics, University of Zurich.

Reinhard, T. and Glinz, M. (2010). Automatic placement of link labels in diagrams. In *Workshop on Flexible Modeling Tools at ICSE'10*, 5 pages.

Reinhard, T., Meier, S., and Glinz, M. (2007). An improved fisheye zoom algorithm for visualizing and editing hierarchical models. In *2nd International Workshop on Requirements Engineering Visualization at RE'07*, 10 pages.

Reinhard, T., Meier, S., Stoiber, R., Cramer, C., and Glinz, M. (2008). Tool support for the navigation in graphical models. In *30th International Conference on Software Engineering*, pages 823–826.

Reinhard, T., Seybold, C., Meier, S., Glinz, M., and Merlo-Schett, N. (2006). Human-friendly line routing for hierarchical diagrams. In *21st International Conference on Automated Software Engineering*, pages 273–276.

Reiser, M.-O. and Weber, M. (2006). Managing highly complex product families with multi-level feature trees. In *14th IEEE International Requirements Engineering Conference*, pages 146–155.

Reiter, R. (1987). A theory of diagnosis from first principles. *Artif. Intell.*, 32:57–95.

RERG, University of Zurich (2011). The ADORA tool. URL: http://www.ifi.uzh.ch/rerg/research/adora/tool.html (checked on July 5, 2012).

Roos-frantz, F., Benavides, D., and Ruiz-Cort, A. (2009). Feature model to orthogonal variability model transformations. A first step. *Actas de los Talleres de las Jornadas de Ing. del Software y BBDD*, 9 pages.

Rosenmüller, M. and Siegmund, N. (2010). Automating the configuration of multi software product lines. In *4th International Workshop on Variability Modeling of Software-intensive Systems*, pages 123–130.

Rosenmüller, M., Siegmund, N., Rahman, S. S. u., and Kästner, C. (2008). Modeling dependent software product lines. In *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering at GPCE'08*, pages 13–18.

Rubin, J. and Chechik, M. (2010). From products to product lines using model matching and refactoring. In *2nd International Workshop on Model-Driven Software Product Line Engineering at SPLC'10*, pages 155–162.

Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified Modeling Language reference manual*. Pearson Higher Education, 2nd edition.

Sánchez, P., Loughran, N., Fuentes, L., and Garcia, A. (2009). Engineering languages for specifying product-derivation processes in software product lines. In *Software Language Engineering*, Springer, pages 188–207.

Schadauer, S. (2011). Industrial evaluation of the SPREBA method to software product line requirements engineering. Master's thesis, Johannes Kepler University Linz.

Schmid, K. and John, I. (2004). A customizable approach to full lifecycle variability management. *Sci. Comput. Program.*, 53(3):259–284.

Schmid, K., Rabiser, R., and Grünbacher, P. (2011). A comparison of decision modeling approaches in product lines. In *5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 119–126.

Schobbens, P.-Y., Heymans, P., and Trigaux, J.-C. (2006). Feature diagrams: a survey and a formal semantics. In *14th International Requirements Engineering Conference*, pages 136–145.

Seybold, C. (2006). *Simulation teilformaler Anforderungsmodelle [Simulation of semi-formal requirements models (in German)]*. PhD thesis, Department of Informatics, University of Zurich.

Seybold, C., Glinz, M., Meier, S., and Merlo-Schett, N. (2003). An effective layout adaptation technique for a graphical modeling tool. In *25th International Conference on Software Engineering*, pages 826–827.

She, S., Lotufo, R., Berger, T., Wąsowski, A., and Czarnecki, K. (2011). Reverse engineering feature models. In *33rd International Conference on Software Engineering*, pages 461–470.

Sheldon, F., Kavi, K., Tausworthe, R., Yu, J., Brettschneider, R., and Everett, W. (1992). Reliability measurement: from theory to practice. *IEEE Software*, 9(4):13 –20.

Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., and Taghdiri, M. (2003). Debugging overconstrained declarative models using unsatisfiable cores. In *18th International Conference on Automated Software Engineering*, pages 94–105.

Sinnema, M. and Deelstra, S. (2007). Classifying variability modeling techniques. *Inf. Softw. Technol.*, 49:717–739.

Software Productivity Consortium Services Corporation (1993). Reuse-driven software processes. Technical report, TR SPC-92019-CMC, Version 02.00.03, Software Productivity Consortium Services Corporation.

Sommerville, I. and Sawyer, P. (1997). *Requirements engineering: a good practice guide*. John Wiley & Sons, 1st edition.

Stachowiak, H. (1973). *Allgemeine Modelltheorie [General model theory (in German)]*. Springer.

Stoiber, R. (2006). Modellierung von Architekturvariabilität in Softwaresystemen [Modeling of architecture variability in software systems (in German)]. Master's thesis, Johannes Kepler University Linz.

Stoiber, R., Fricker, S., Jehle, M., and Glinz, M. (2010). Feature unweaving: refactoring software requirements specifications into software product lines. In *18th International Requirements Engineering Conference*, pages 403–404.

Stoiber, R. and Glinz, M. (2009). Modeling and managing tacit product line requirements knowledge. In *2nd International Workshop on Managing Requirements Engineering Knowledge at RE'09*, pages 60–64

Stoiber, R. and Glinz, M. (2010a). Feature unweaving: efficient variability extraction and specification for emerging software product lines. In *4th International Workshop on Software Product Management at RE'10*, pages 53–62.

Stoiber, R. and Glinz, M. (2010b). Supporting stepwise, incremental product derivation in product line requirements engineering. In *4th International Workshop on Variability Modelling of Software-intensive Systems*, pages 77–84.

Stoiber, R., Meier, S., and Glinz, M. (2007). Visualizing product line domain variability by aspect-oriented modeling. In *2nd International Workshop on Requirements Engineering Visualization at RE'07*, pages 59–64.

Stoiber, R., Reinhard, T., and Glinz, M. (2008). Visualization support for software product line modeling. In *2nd International Workshop on Visualisation in Software Product Line Engineering at SPLC'08*, pages 313–322.

Storey, M.-A. D. and Müller, H. A. (1996). Graph layout adjustment strategies. In *Symposium on Graph Drawing*, pages 487–499.

Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr., S. M. (1999). N degrees of separation: multi-dimensional separation of concerns. In *21st International Conference on Software Engineering*, pages 107–119.

Thüm, T., Batory, D., and Kästner, C. (2009). Reasoning about edits to feature models. In *31st International Conference on Software Engineering*, pages 254–264.

Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A., and Toro, M. (2008). Automated error analysis for the agilization of feature modeling. *J. Syst. Softw.*, 81:883–896.

van Lamsweerde, A. (2009). *Requirements engineering: from system goals to UML models to software specifications*. John Wiley & Sons.

van Ommering, R. (2002). Building product populations with software components. In *24th International Conference on Software Engineering*, pages 255–265.

Voelter, M. (2010). Implementing feature variability for models and code with projectional language workbenches. In *2nd International Workshop on Feature-Oriented Software Development at GPCE and SLE '10*, pages 41–48.

von der Maßen, T. (2007). *Feature-basierte Modellierung und Analyse von Variabilität in Produktlinienanforderungen [Feature-based modeling and analysis of variability in product line requirements (in German)]*. PhD thesis, RWTH Aachen.

Wang, B., Hu, Z., Xiong, Y., Zhao, H., Zhang, W., and Mei, H. (2010). Tolerating inconsistency in feature models. In *3rd Workshop on Living With Inconsistency in Software Development at ASE'10*, pages 15-20.

Wang, B., Zhang, W., Zhao, H., Jin, Z., and Mei, H. (2009). A use case based approach to feature models' construction. In *17th International Requirements Engineering Conference*, pages 121–130.

Weiss, D. M. and Lai, C. T. R. (1999). *Software product-line engineering: a family-based software development process*. Addison-Wesley.

Welzl, E. (2005). Boolean satisfiability – combinatorics and algorithms. Course notes, ETH Zurich, URL: http://www.inf.ethz.ch/ emo/SmallPieces/SAT.ps (checked on July 5, 2012),

Weston, N., Chitchyan, R., and Rashid, A. (2009). A framework for constructing semantically composable feature models from natural language requirements. In *13th International Software Product Line Conference*, pages 211–220.

White, J., Benavides, D., Dougherty, B., and Schmidt, D. C. (2009). Automated reasoning for multi-step feature model configuration problems. In *13th International Software Product Line Conference*, pages 11–20.

White, J., Schmidt, D. C., Benavides, D., Trinidad, P., and Ruiz-Cortés, A. (2008). Automated diagnosis of product-line configuration errors in feature models. In *12th International Software Product Line Conference*, pages 225–234.

Whittle, J. and Jayaraman, P. K. (2007). MATA: a tool for aspect-oriented modeling based on graph transformation. In *11th International Workshop on Aspect-Oriented Modeling at MoDELS '07*, pages 16–27.

Whittle, J., Jayaraman, P. K., Elkhodary, A. M., Moreira, A., and Araújo, J. (2009). MATA: a unified approach for composing UML aspect models based on graph transformation. *Transactions on Aspect-Oriented Software Development VI*, 6:191–237.

Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20:822–823.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers.

Xia, Y. (2005). *A language definition method for visual specification languages*. PhD thesis, Department of Informatics, University of Zurich.

Xiong, Y., Hubaux, A., She, S., and Czarnecki, K. (2011). Generating range fixes for software configuration. Technical report, GSDLAB-TR 2011-10-27, University of Waterloo.

Yin, R. K. (2002). *Case study research: design and method.* Vol. 5 of *Applied Social Research Methods Series*, Sage Publications, 3rd edition.

Zhang, W., Zhao, H., and Mei, H. (2004). A propositional logic-based method for verification of feature models. In *Formal Methods and Software Engineering*, pages 115–130.

Zhang, X., Haugen, O., and Moller-Pedersen, B. (2011). Model comparison to synthesize a model-driven software product line. In *15th International Software Product Line Conference*, pages 90–99.

Ziadi, T. and Jézéquel, J.-M. (2006). Software product line engineering with the UML: deriving products. In *Software Product Lines*, pages 557–588.

Zoller, U. (2010). A comparison of three different concepts for requirements modeling of software product lines – a case study-based investigation (in German). Bachelor's thesis, Department of Informatics, University of Zurich.

Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., and Kulesza, U. (2009). VML* - a family of languages for variability management in software product lines. In *2nd International Conference on Software Language Engineering*, pages 82–102.

Curriculum Vitae

Name:	Reinhard Stoiber
Date and Place of Birth:	April 10, 1982 in Wels, Upper Austria
Nationality:	Austrian

2012	Ph.D. thesis submitted on January 19 at University of Zurich, Switzerland
2006-2011	Assistant and doctoral student at the Department of Informatics, University of Zurich, Switzerland
2005-2006	Diploma thesis in cooperation with Siemens VAI and graduation to Mag. rer. soc. oec. at Johannes Kepler University Linz, Austria
2006	Civilian service at assista Soziale Dienste GmbH, Austria
2004-2005	Erasmus exchange student at University of Tampere, Finland
2001-2006	Academic study of business informatics at Johannes Kepler University Linz, Austria
1996-2001	High school (upper secondary technical and vocational college) with focus on electronic engineering at HTL Leonding, Austria